# A UNIFIED PEER-TO-PEER DATABASE FRAMEWORK FOR XQUERIES OVER DYNAMIC DISTRIBUTED CONTENT AND ITS APPLICATION FOR SCALABLE SERVICE DISCOVERY

**DISSERTATION**

**DIPL. ING. WOLFGANG HOSCHEK**

AUSGEFÜHRT ZUM ZWECKE DER ERLANGUNG DES
AKADEMISCHEN GRADES EINES
DOKTORS DER TECHNISCHEN WISSENSCHAFTEN

EINGEREICHT AN DER TECHNISCHEN UNIVERSITÄT WIEN
FAKULTÄT FÜR TECHNISCHE NATURWISSENSCHAFTEN UND
INFORMATIK

UNTER DER ANLEITUNG VON
O.UNIV.-PROF. DIPL.-ING. MAG. DR. GERTI KAPPEL
AO.UNIV.-PROF. DR. ERICH SCHIKUTA
CERN BETREUER: DR. BERND PANZER-STEINDEL

GENF, IM MÄRZ 2002

# Contents

# Abstract

In a large distributed system spanning administrative domains such as a Grid, it is desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. The *web services* vision promises that programs are made more flexible and powerful by querying Internet databases (registries) at runtime in order to discover information and network attached third-party building blocks. Services can advertise themselves and related metadata via such databases, enabling the assembly of distributed higher-level components. In support of this vision, this thesis shows how to support expressive general-purpose queries over a view that integrates autonomous dynamic database nodes from a wide range of distributed system topologies.

We motivate and justify the assertion that realistic ubiquitous service and resource discovery requires a rich general-purpose query language such as XQuery or SQL. Next, we introduce the *Web Service Discovery Architecture (WSDA)*, which subsumes an array of disparate concepts, interfaces and protocols under a single semi-transparent umbrella. WSDA specifies a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery. These primitives cover service identification, service description retrieval, data publication as well as minimal and powerful query support. The individual primitives can be combined and plugged together by specific clients and services to yield a wide range of behaviors and emerging synergies. Based on WSDA, we introduce the *hyper registry*, which is a centralized database node for discovery of dynamic distributed content, supporting XQueries over a tuple set from an XML data model. We address the problem of maintaining dynamic and timely information populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources.

However, in a large cross-organizational system, the set of information tuples is partitioned over many such distributed nodes, for reasons including autonomy, scalability, availability, performance and security. This suggests the use of Peer-to-Peer (P2P) query technology. Consequently, we take the first steps towards unifying the fields of database management systems and P2P computing. As a result, we propose the WSDA based *Unified Peer-to-Peer Database Framework (UPDF)* and its associated *Peer Database Protocol (PDP)*, which are unified in the sense that they allow to express specific applications for a wide range of data types (typed or untyped XML, any MIME type), node topologies (e.g. ring, tree, graph), query languages (e.g. XQuery, SQL), query response modes (e.g. Routed, Direct and Referral Response), neighbor selection policies, pipelining, timeout and other scope characteristics.

The uniformity and wide applicability of our approach is distinguished from related work, which (1) addresses some but not all problems, and (2) does not propose a unified framework.

# Zusammenfassung

In einem mehrere Organisationen überspannenden, großen verteilten System, wie z.B. einem Grid, ist es wünschenswert dynamische und zeitsensitive Information über Netzwerkdienste, Ressourcen und Benutzer zu verwalten und abzufragen. Das Konzept der Webdienste verspricht flexible Programme die zur Laufzeit Internet Datenbanken (Registries) benutzen um Informationen und Netzwerkdienste von Drittanbietern zu finden. Dienste können sich und verwandte Metadaten durch derartige Datenbanken anbieten und so das Zusammenfügen von höheren verteilten Komponenten ermöglichen. Diese Dissertation unterstützt diese Vision indem sie zeigt, wie ausdrucksstarke Mehrzweckabfragen ueber eine Sicht formuliert werden können, die autonome dynamische Datenbankknoten von beliebigen Topologien integriert.

Wir motivieren und rechtfertigen die Behauptung, daß das Finden von Ressourcen und Diensten eine reiche Mehrzweckabfragesprache wie z.B. XQuery oder SQL verlangt. Wir führen die sogenannte *Web Service Discovery Architecture (WSDA)* ein, die disparate Konzepte, Schnittstellen und Netzwerkprotokolle unter einem quasi-transparenten Dach zusammenfaßt. WSDA spezifiziert eine kleine Menge von orthogonalen Mehrzweckfunktionen (Bausteinen) zum Finden von Diensten. Diese decken die Bereiche der Dienstidentifizierung, Dienstbeschreibung, Datenpublikation sowie minimale und mächtige Abfrageunterstützung ab. Clients und Server können diese Funktionen so kombinieren daß dabei eine breite Palette von Verhalten und Synergien ensteht. Basierend auf WSDA führen wir eine zentrale Datenbank für das Finden von dynamischen verteilten Daten ein, die *Hyper Registry*. Diese unterstützt XQueries über dynamische, zeitsensitive Daten, die von unzuverlässigen, sich häufig ändernden, autonomen und heterogenen Datenquellen stammen.

In einem mehrere Organisationen überspannenden, großen verteilten System jedoch sind die Datentupel über viele Knoten verteilt, z.B. aus Gründen der Autonomie, Skalierbarkeit, Verfügbarkeit, Effizienz und Sicherheit. Daher empfiehlt sich die Benützung von Peer-to-Peer (P2P) Abfragetechnologie. So unternehmen wir die ersten Schritte zur Vereinheitlichung von Datenbank Management Systemen und P2P Computing. Auf WSDA basierend schlagen wir das *Unified Peer-to-Peer Database Framework (UPDF)* und korrespondierende *Peer Database Protocol (PDP)* vor. Beide sind vereinheitlicht in dem Sinn daß innerhalb ihreres Rahmens spezifische Applikationen für eine Vielfalt von Datentypen, Knotentopologien, Abfragesprachen, Antwort-Modi, und verschiedene Formen der Nachbarschaftsauswahl, des Pipelining, und von Timeouts formuliert werden können.

Die Einheitlichkeit, breite Einsatzfähigkeit und Wiederverwendbarkeit unseres Ansatzes unterscheidet sich von verwandten Arbeiten die (1) einzelne aber nicht alle Probleme behandeln, und (2) keinen einheitlichen Rahmen einführen.

# Chapter 1

# Introduction

## 1.1 Motivation

This thesis tackles the problems of information, resource and service discovery arising in large distributed Internet systems spanning multiple administrative domains. We show how to support expressive general-purpose queries over a view that integrates autonomous dynamic database nodes from a wide range of distributed system topologies. The work was carried out in the context of the European DataGrid project (EDG) [1, 2, 3] at CERN, the European Organization for Nuclear Research, and supported by the Austrian Ministerium für Wissenschaft, Bildung und Kultur.

The international High Energy Physics (HEP) research community is facing a substantial challenge in joining a massive set of loosely coupled people and resources from multiple distributed organizations. Although this is the driving use case of the EDG project, this thesis distills and generalizes the essential properties of the discovery problem and then develops generic solutions that apply to a wide range of large distributed Internet systems. However, before we do so, it is helpful to introduce as an example the scenario of a HEP DataGrid Collaboration.

The HEP community comprises thousands of researchers globally spread over hundreds of laboratories and university departments. These include CERN (International); Stanford, Berkeley, Caltech, Fermilab (USA); INFN (Italy); RAL (UK); IN2P3 (France), and more organizations from some 80 countries such as Austria, Brazil, China, Germany, Greece, India, Japan, Pakistan, Mexico, Portugal, Russia, Spain, Turkey, etc. What is unique is that these dispersed researchers *jointly* undertake so-called physics *experiments*, which are mammoth projects with a lifetime of years or sometimes decades. These experiments help to study the fundamental forces of nature, such as the origin of gravitation.

To conduct an experiment, machines called particle accelerators and detectors are constructed. An accelerator typically consists of a several kilometer-long circular underground vacuum tube surrounded by super-conducting magnets. Subatomic particles such as protons and electrons are generated, focused into beams and injected into the accelerator. Typically, two slightly offset beams concurrently travel the tube in opposite directions. Large oscillating magnets pull and push the particles, accelerating them to near light speed. The circling beams are steered into a particle detector where they are made to collide head on, causing the quantum mechanical equivalent of a car crash where opposite highway lanes are directed into each other (Figure 1.1). For a brief moment in time, very high energies are concentrated

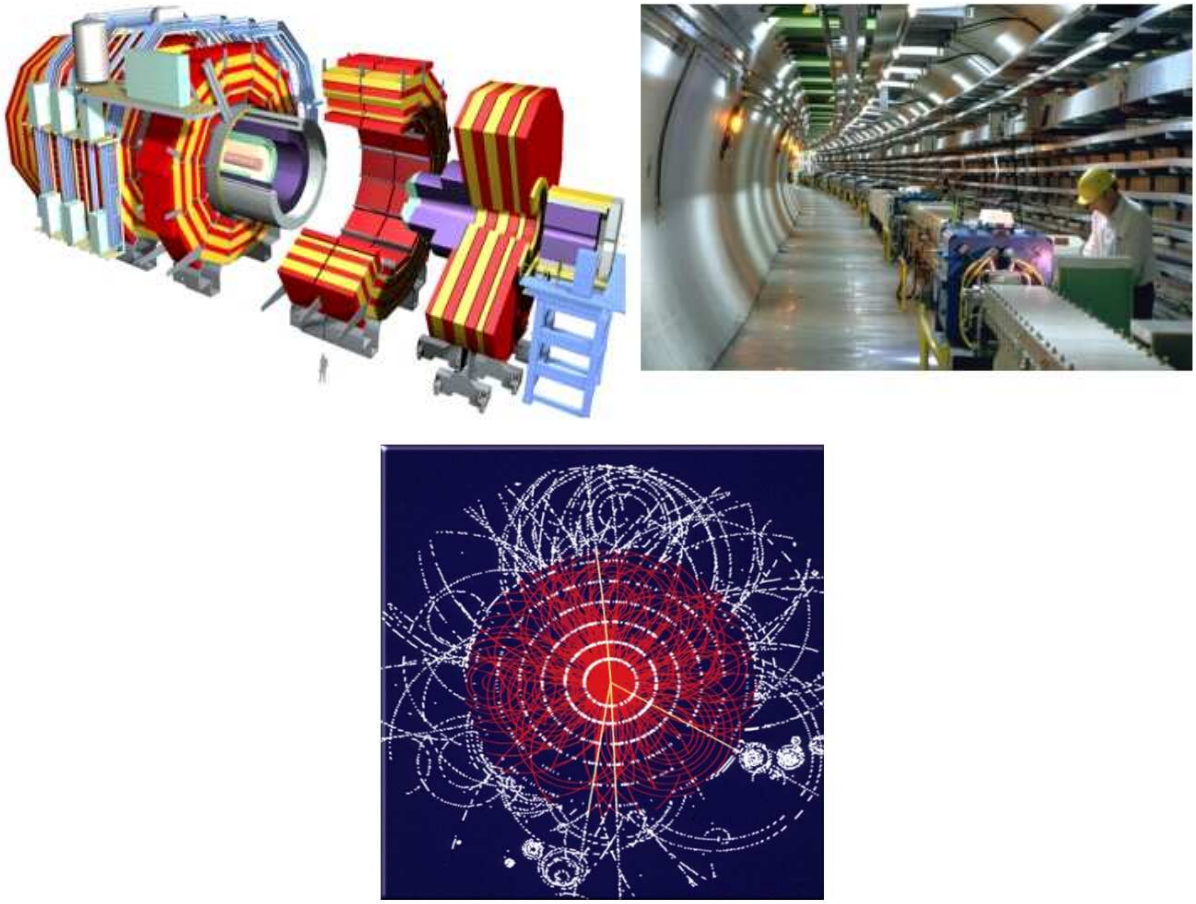

Figure 1.1: Particle Detector (top left), Accelerator (top right) and Collision (bottom).

in a small space, forming the conditions under which *strange things* happen, including cascading creation and decay of particles and their byproducts, and the birth of exotic particles believed to have existed only shortly after the Big Bang, the creation of the universe. In effect, detectors are the equivalent of cameras positioned to the left and right of the highway, collecting incomplete car crash evidence from which, with good detective work and a bit of luck, one can infer just what has happened. Data acquisition systems record the traces of collision events and transform them into digital data. The collision data continually arrives at rates around 100 - 1000 MB/sec. It is streamed via networks into a computer center nearby, where it is pre-filtered in real-time in so-called event filter farms, which are large dedicated computer clusters with high bandwidth low latency interconnects. The pre-filtered data is then stored in robotic tape libraries and disk arrays for later analysis.

Physics data analysis is a complex, tedious and slow iterative process, carried out in several stages, during each of which Gigabytes, Terabytes and soon Petabytes of data are filtered, transformed and distilled. Analysis continues until, eventually, and with luck, a plot visualizes a bizarre and currently unexplainable physics phenomenon, based on which a Nobel

Figure 1.2: CERN's Users in the World.

Prize may or may not be granted some 20 years later.

A massive set of computing resources is necessary for CERN's next generation Large Hadron Collider (LHC) project, including thousands of network services, tens of thousands of CPUs, WAN Gigabit networking as well as Petabytes of disk and tape storage [4]. The cost of the infrastructure necessary to support HEP experiments and their data analysis is beyond the financial capabilities of individual universities and even states. Hence, in the past, resources have been concentrated at a handful of laboratories, primarily in Switzerland, USA, Japan and Germany. CERN, the European Organization for Nuclear Research, straddling the Franco-Swiss border near Geneva, is the largest such laboratory. Some 3000 staff and 6500 scientists from 500 universities, half of the world's particle physicists, use CERN's facilities (Figure 1.2). Its annual budget of 600 million US-$ is financed by 20 member states and some 30 associate states.

The LHC project requires unprecedented computing resources, and involves substantially more remote researchers and institutions than any prior project. To make their collaboration viable, it was decided to share in a global joint effort the data and locally available resources of all participating laboratories and university departments. Because the technology necessary to achieve this goal was only partially available at the time, the European DataGrid project (EDG) was born. Relevant parts of the detector data are distributed from CERN to associate laboratories, organized into a multi-tier architecture sketched in Figure 1.3.

The fundamental value proposition of computer systems has long been their potential to automate well-defined repetitive tasks. With the advent of distributed computing, the Internet and web technologies in particular, the focus has been broadened. Increasingly, computer systems are seen as enabling tools for effective long distance communication and collaboration. Colleagues (and programs) with shared interests can better work together, with less respect to the physical location of themselves and required devices and machinery. The traditional departmental team is complemented by cross-organizational virtual teams, operating in an open, transparent manner. Such teams have been termed *virtual organizations*

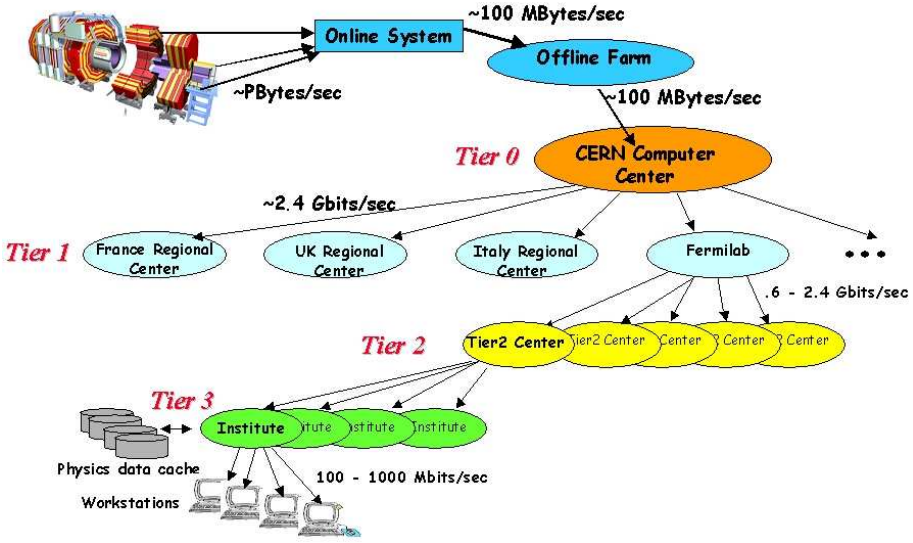


Figure 1.3: Multi-Tier Architecture of European DataGrid.

[5]. This opportunity to further extend knowledge appears natural to science communities since they have a deep tradition in drawing their strength from stimulating partnerships across administrative boundaries.

Grid technology attempts to support flexible, secure, coordinated information sharing among dynamic collections of individuals, institutions and resources. This includes data sharing but also includes access to computers, software and devices required by computation and data-rich collaborative problem solving [5]. These and other advances of distributed computing are necessary to increasingly make it possible to join loosely coupled people and resources from multiple organizations.

As a result, many entities can now collaborate among each other to enable data analysis of large HEP experimental data: the HEP user community and its multitude of institutions, storage providers, as well as network, application and cycle providers. Users utilize the services of a set of remote application providers to submit jobs, which in turn are executed by the services of cycle providers, using storage and network provider services for I/O. The services necessary to execute a given task often do not reside in the same administrative domain. Collaborations may have a rather static configuration, or they may be more dynamic and fluid, with users and service providers joining and leaving frequently, and configurations as well as usage policies often changing.

## 1.2 Background

Component oriented software development has advanced to a state where a large fraction of the functionality required for typical applications is available through third party libraries, frameworks and tools. These components are often reliable, well documented and maintained, and designed with the intention to be reused and customized. For many software developers

the key skill is no longer hard core programming, but rather the ability to find, assess and integrate building blocks from a large variety of third parties.

The software industry has steadily moved towards more software execution flexibility. For example, dynamic linking allows for easier customization and upgrade of applications than static linking. Modern programming languages such as Java use an even more flexible link model that delays linking until the last possible moment (the time of method invocation). Still, most software expects to link and run against third party functionality installed on the local computer executing the program. For example, a word processor is locally installed together with all its internal building blocks such as spell checker, translator, thesaurus and modules for import and export of various data formats. The network is not an integral part of the software execution model, whereas the local disk and operating system certainly are.

The maturing of Internet technologies has brought increased ease-of-use and abstraction through higher-level protocol stacks, improved APIs, more modular and reusable server frameworks and correspondingly powerful tools. The way is now paved for the next step towards increased software execution flexibility. In this scenario, some components are network-attached and made available in the form of network *services* for use by the general public, collaborators or commercial customers. Internet Service Providers (ISPs) offer to run and maintain reliable services on behalf of clients through hosting environments. Rather than invoking functions of a local library, the application now invokes functions on remote components, in the ideal case to the same effect.

Remote invocation is always necessary for some demanding applications that cannot (exclusively) be run locally on the computer of a user because they depend on a set of resources scattered over multiple remote domains. Examples include computationally demanding gene sequencing, business forecasting, climate change simulation and astronomical sky surveying as well as data-intensive High Energy Physics analysis sweeping over Terabytes of data. Such applications can reasonably only be run on a remote supercomputer or several large computing clusters with massive CPU, network, disk and tape capacities, as well as an appropriate software environment matching minimum standards.

The most straightforward but also most inflexible configuration approach is to hard wire the location, interface, behavior and other properties of remote services into the local application. Loosely coupled decentralized systems call for solutions that are more flexible and can seamlessly adapt to changing conditions. For example, if a user turns out to be less than happy with the perceived quality of a word processor's remote spell checker, he/she may want to plug in another spell checker. Such dynamic plug-ability may become feasible if service implementations adhere to some common interfaces and network protocols, and if it is possible to match services against an interface and network protocol specification. An interesting question then is: *What infrastructure is necessary to enable a program to have the capability to search the Internet for alternative but similar services and dynamically substitute these?*

Consequently, the next step towards increased execution flexibility is the (still immature and hence often hyped) *web services* vision [6, 7] of distributed computing where programs are no longer configured with static information. Rather, the promise is that programs are made more flexible and powerful by querying Internet databases (registries) at runtime in order to discover information and network attached third-party building blocks. Services can advertise themselves and related metadata via such databases, enabling the assembly of

distributed higher-level components. A natural question arises. *How precisely can a local application discover relevant remote services?*

For example, a data-intensive High Energy Physics analysis application looks for remote services that exhibit a suitable combination of characteristics, including network load, available disk quota, security options, access rights, and perhaps Quality of Service and monetary cost. What is more, it is often necessary to use several services in combination to implement the operations of a request. For example, a request may involve the combined use of a file transfer service (to stage input and output data from remote sites), a replica catalog service (to locate an input file replica with good data locality), a request execution service (to run the analysis program) and finally again a file transfer service (to stage output data back to the user desktop). In such cases it is often helpful to consider correlations. For example, a scheduler for data-intensive requests may look for input file replica locations with a fast network path to the execution service where the request would consume the input data. If a request involves reading large amounts of input data, it may be a poor choice to use a host for execution that has poor data locality with respect to an input data source, even if it is very lightly loaded. *How can one find a set of correlated services fitting a complex pattern of requirements and preferences?*

If one instance of a service can be made available, a natural next step is to have more than one identical distributed instance, for example to improve availability and performance. Changing conditions in distributed systems include latency, bandwidth, availability, location, access rights, monetary cost and personal preferences. For example, adaptive users or programs may want to choose a particular instance of a content download service depending on estimated download bandwidth. If bandwidth is degraded in the middle of a download, a user may want to switch transparently to another download service and continue where he/she left off. *On what basis could one discriminate between several instances of the same service?*

In a large heterogeneous distributed system spanning multiple administrative domains, it is desirable to maintain and query dynamic and timely information about the active participants such as services, resources and user communities. Examples are a (worldwide) service discovery infrastructure for a DataGrid, the Domain Name System (DNS), the email infrastructure, the World Wide Web, a monitoring infrastructure or an instant news service. However, the set of information tuples in the universe is partitioned over one or more database nodes from a wide range of system topologies, for reasons including autonomy, scalability, availability, performance and security. The goal is to exploit several independent information sources as if they were a single source. This enables queries for information, resource and service discovery and collective collaborative functionality that operate on the system as a whole, rather than on a given part of it. For example, it allows a search for descriptions of services of a file sharing system, to determine its total download capacity, the names of all participating organizations, etc.

However, in such large distributed systems it is hard to keep track of metadata describing participants such as services, resources, user communities and data sources. Predictable, timely, consistent and reliable global state maintenance is infeasible. The information to be aggregated and integrated may be outdated, inconsistent, or not available at all. Failure, misbehavior, security restrictions and continuous change are the norm rather than the

exception. Consider an instant news service that aggregates news from a large variety of autonomous remote data sources residing within multiple administrative domains. New data sources are being integrated frequently and obsolete ones are dropped. One cannot force control over multiple administrative domains. Reconfiguration or physical moving of a data source is the norm rather than the exception. The question then is: *How can one keep track of and query the metadata describing the participants of large cross-organizational distributed systems undergoing frequent change?*

## 1.3 Contribution and Organization of this Thesis

This thesis addresses the following open problems:

- In a distributed system, it is desirable to maintain dynamic and timely information about active participants such as services, resources and user communities. The web service vision promises to make programs more flexible and powerful by consulting Internet databases (registries) at runtime in order to discover information and network attached third-party building blocks. While important advances have recently been made in the field of web service specification [8], invocation [9] and registration [10], the problem of how to use a rich and expressive general-purpose query language to discover services that offer functionality matching a detailed specification has so far received little attention.

- In a large distributed system spanning many administrative domains, the set of information tuples in the universe is partitioned over one or more database nodes from a wide range of system topologies, for reasons including autonomy, scalability, availability, performance and security. Each node is populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources. Under such conditions, predictable, timely, consistent and reliable global state maintenance is infeasible. The problem of how to support expressive general-purpose discovery queries over a view that integrates autonomous dynamic database nodes from a wide range of distributed system topologies has so far not been addressed.

This thesis is organized into chapters as follows:

**Chapter 2.** The most straightforward but also most inflexible configuration approach for invocation of remote services is to hard wire the location, interface, behavior and other properties of remote services into the local application. Loosely coupled decentralized systems call for solutions that are more flexible and can seamlessly adapt to changing conditions. A key question then is:

- *What distinct problem areas and processing steps can be distinguished in order to enable flexible remote invocation in the context of service discovery?*

**Chapter 2 – Contribution.** To establish the context, we outline eight problem areas and their associated processing steps, namely *description, presentation, publication, request, discovery, brokering, execution* and *control*. We propose a simple grammar (*SWSDL*) for describing network services as collections of service interfaces capable of executing operations over network protocols to endpoints. The grammar is intended to be used in the high-level architecture and design phase of a software project. A service must present its current description so that clients from anywhere can retrieve it at any time. For broad acceptance, adoption and easy integration of legacy services, an HTTP hyperlink is chosen as an identifier and retrieval mechanism (*service link)*. A registry for publication and query of service and resource presence information is outlined. Reliable, predictable and simple distributed registry state maintenance in the presence of service failure, misbehavior or change is addressed by a simple and effective soft state mechanism. The notions of *request, resource* and *operation* are clarified. We outline the discovery step, which finds services implementing the operations required by a request. The brokering step determines an invocation schedule, which is a mapping over time of unbound operations to service operation invocations using given resources. The execution step implements a schedule. It uses the supported protocols to invoke operations on remote services. We discuss how one can reliably support monitoring and controlling the lifecycle of a request in the presence of a service that cannot reliably complete a request within a short and well-known expected timeframe.

**Chapter 3.** In a large cross-organizational distributed system, the set of information tuples in the universe is partitioned over one or more database nodes from a wide range of distributed system topologies, for reasons including autonomy, scalability, availability, performance and security. A database model is required that clarifies the relationship of the entities in a distributed system.

The distribution and location of tuples should be transparent to a query. However, in practice, it is often sufficient (and much more efficient) for a query to consider only a subset of all tuples (service descriptions) from a subset of nodes. For example, a typical query may only want to search tuples (services) within the scope of the domain `cern.ch` and ignore the rest of the world. Both requirements need to be addressed by an appropriate query model.

A data model remains to be specified. It should specify what kind of data a query takes as input and produces as output. Due to the heterogeneity of large distributed systems, the data model should be flexible in representing many different kinds of information from diverse sources, including structured and semi-structured data. The key problem then is:

- *What kind of database, query and data model as well as query language can support simple and complex dynamic information discovery with as few as possible architecture and design assumptions? In particular, how can one uniformly support queries in a wide range of distributed system topologies and deployment models, while at the same time accounting for their respective characteristics?*

**Chapter 3 - Contribution.** The chapter develops a database and query model as well as a generic and dynamic data model that address the given problem. All subsequent chapters are based on these models. Unlike in the relational model the elements of a tuple in our

data model can hold structured or semi-structured data in the form of any arbitrary well-formed XML [11] document or fragment. An individual tuple element may, but need not, have a schema (XML Schema [12]), in which case the element must be valid according to the schema. The elements of all tuples may, but need not, share a common schema. The concepts of (logical) query and (physical) query scope are cleanly separated rather than interwoven. A query is formulated against a global database view and is insensitive to link topology and deployment model. In other words, to a query the set of all tuples appears as a single homogenous database, even though the set may be (recursively) partitioned across many nodes and databases. The query scope, on the other hand, is used to navigate and prune the link topology and filter on attributes of the deployment model. A query is evaluated against a set of tuples. The set, in turn, is specified by the scope. Conceptually, the scope is the input fed to the query. Example service discovery queries are given. Three query types are identified, namely *simple, medium* and *complex*. An appropriate query language (XQuery) is suggested. The suitability of the query language is demonstrated by formulating the example prose queries in the language. Detailed requirements for a query language supporting service and resource discovery are given. The capabilities of various query languages are compared.

**Chapter 4.** In a large distributed system, a variety of information describes the state of autonomous entities from multiple administrative domains. Participants frequently join, leave and act on a best effort basis. Predictable, timely, consistent and reliable global state maintenance is infeasible. The information to be aggregated and integrated may be outdated, inconsistent, or not available at all. Failure, misbehavior, security restrictions and continuous change are the norm rather than the exception. The key problem then is:

- *How should a database node maintain information populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources? In particular, how should it do so without sacrificing reliability, predictability and simplicity? How can powerful queries be expressed over time-sensitive dynamic information?*

**Chapter 4 - Contribution.** A type of database is developed that addresses the problem. A database for XQueries over dynamic distributed content is designed and specified – the so-called *hyper registry*. The hyper registry has a number of key properties. An XML data model allows for structured and semi-structured data, which is important for integration of heterogeneous content. The XQuery language allows for powerful searching, which is critical for non-trivial applications. Database state maintenance is based on soft state, which enables reliable, predictable and simple content integration from a large number of autonomous distributed content providers. Content link, content cache and a hybrid pull/push communication model allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client.

These key properties distinguish our approach from related work, which individually addresses some, but not all of the above issues. Some work does not follow an XML data model (X.500 [13], LDAP [14], MDS [15, 16], RDBMS). Sometimes the query language is not powerful enough (UDDI [10], X.500, LDAP, MDS). Sometimes the database is not based

on soft state (RDBMS, UDDI, X.500, LDAP). Sometimes content freshness is not addressed (RDBMS, UDDI, X.500, LDAP) or only partly addressed (MDS).

**Chapter 5.** Having defined all registry aspects in detail, we now proceed to the definition of a web service layer that promotes interoperability for existing and future Internet software. Such a layer views the Internet as a large set of services with an extensible set of well-defined interfaces. A web service consists of a set of interfaces with associated operations. Each operation may be bound to one or more network protocols and endpoints. The definition of interfaces, operations and bindings to network protocols and endpoints is given as a service description. A discovery architecture defines appropriate services, interfaces, operations and protocol bindings for discovery. The key problem is:

- *Can we define a discovery architecture that promotes interoperability, embraces industry standards, and is open, modular, flexible, unified, non-intrusive and simple yet powerful?*

**Chapter 5 - Contribution.** We propose and specify such a discovery architecture, the so-called *Web Service Discovery Architecture (WSDA)*. WSDA subsumes an array of disparate concepts, interfaces and protocols under a single semi-transparent umbrella. It specifies a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery. These primitives cover service identification, service description retrieval, data publication as well as minimal and powerful query support. The individual primitives can be combined and plugged together by specific clients and services to yield a wide range of behaviors and emerging synergies. Finally, we compare in detail the properties of WSDA with the emerging Open Grid Service Architecture [6, 17].

**Chapter 6.** Because the set of information tuples in the universe is partitioned over one or more distributed nodes, it is not obvious how to enable powerful discovery query support and collective collaborative functionality that operate on the distributed system as a whole, rather than on a given part of it. Further, it is not obvious how to allow for search results that are fresh, allowing dynamic content. It appears that a Peer-to-Peer (P2P) database network may be well suited to support dynamic distributed database search, for example for service discovery. The key problems then are:

- *What are the detailed architecture and design options for P2P database searching in the context of service discovery? What response models can be used to return matching query results? How should a P2P query processor be organized? What query types can be answered (efficiently) by a P2P network? What query types have the potential to immediately start piping in (early) results? How can a maximum of results be delivered reliably within the time frame desired by a user, even if a query type does not support pipelining? How can loops be detected reliably using timeouts? How can a query scope be used to exploit topology characteristics in answering a query? For improved efficiency, how can queries be executed in containers that concentrate distributed P2P database nodes into hosting environments with virtual nodes?*

- *Can we devise a unified P2P database framework for general-purpose query support in large heterogeneous distributed systems spanning many administrative domains? More precisely, can we devise a framework that is unified in the sense that it allows to express specific applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options?*

**Chapter 6 - Contribution.** We take the first steps towards unifying the fields of database management systems and P2P computing, which so far have received considerable, but separate, attention. We extend database concepts and practice to cover P2P search. Similarly, we extend P2P concepts and practice to support powerful general-purpose query languages such as XQuery [18] and SQL [19]. As a result, we propose the so-called *Unified Peer-to-Peer Database Framework (UPDF)* for general-purpose query support in large heterogeneous distributed systems spanning many administrative domains. UPDF is unified in the sense that it allows to express specific applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options.

The uniformity, wide applicability and reusability of our approach distinguish it from related work, which individually addresses some but not all problem areas. Traditional distributed systems assume a particular type of topology (e.g. hierarchical as in DNS [20], LDAP [14]). P2P systems are built for a single application and data type and do not support queries from a general-purpose query language. For example, DNS, Gnutella [21], Freenet [22], Tapestry [23], Chord [24] and Globe [25] only support lookup by key (e.g. globally unique name). Others such as SDS [26], LDAP [14] and MDS [15, 16] support simple special-purpose query languages, leading to special-purpose solutions unsuitable for multi-purpose service and resource discovery in large heterogeneous distributed systems spanning many administrative domains. [27, 28, 29] discuss in isolation select neighbor selection techniques for a particular query type, without the context of a framework for query support. LDAP and MDS do not support essential features for P2P systems such as reliable loop detection, non-hierarchical topologies, dynamic abort timeout, query pipelining across nodes as well as radius scoping. None introduce a unified P2P database framework for general-purpose query support.

**Chapter 7.** We describe how the operations of the P2P database framework (UPDF) and registry `XQuery` interface from Section 5.2 are carried over (bound to) a network protocol. A messaging model and network protocol describes the interactions between framework nodes. For large distributed systems, it strongly influences system properties such as reliability, efficiency, scalability, complexity, interoperability, extensibility and, of course, limitations in applicability. The key problem then is:

- *What messaging and communication model, as well as network protocol, uniformly support P2P database queries for a wide range of database architectures and response models such that the stringent demands of ubiquitous Internet discovery infrastructures in terms of scalability, efficiency, interoperability, extensibility and reliability can be met?*

*In particular, how can one allow for high concurrency, low latency as well as early and/or partial result set retrieval? How can one encourage resource consumption and flow control on a per query basis?*

**Chapter 7 - Contribution.** These problems are addressed by developing a suitable messaging, communication and network protocol model, collectively termed *Peer Database Protocol (PDP)*. PDP has a number of key properties. It is applicable to any node topology (e.g. centralized, distributed or P2P) and to multiple P2P response modes (routed response and direct response, both with and without metadata modes). To support loosely coupled autonomous Internet infrastructures, the model is connection-oriented (ordered, reliable, congestion sensitive) and message-oriented (loosely coupled, operating on structured data). For efficiency, it is stateful at the protocol level, with a transaction consisting of one or more discrete message exchanges related to the same query. It allows for low latency, pipelining, early and/or partial result set retrieval due to synchronous pull, and result set delivery in one or more variable sized batches. It is efficient, due to asynchronous push with delivery of multiple results per batch. It provides for resource consumption and flow control on a per query basis, due to the use of a distinct channel per transaction. It is scalable, due to application multiplexing, which allows for very high query concurrency and very low latency, even in the presence of secure TCP connections. To encourage interoperability and extensibility it is fully based on Internet Engineering Task Force (IETF) standards, for example in terms of asynchrony, encoding, framing, authentication, privacy and reporting.

These key properties distinguish our approach from related work, which individually address some, but not all of the above issues. We are not aware of related work that proposes a uniform messaging model applicable to any node topology and at the same time to multiple P2P response modes. Some related work does not apply to loosely coupled database nodes (RDBMS). Some protocols are not stateful at the protocol level (HTTP based mechanisms). Some do not support synchronous pull (LDAP, MDS, Gnutella, Freenet) and result set delivery in one or more variable sized batches (LDAP, MDS, HTTP based mechanisms). Some do not support asynchronous push with delivery of multiple results per batch (LDAP, MDS, HTTP based mechanisms). Some do not provide for resource consumption and flow control on a per query basis (LDAP, MDS, Gnutella, Freenet, HTTP based mechanisms). Some lack application multiplexing for scalable query concurrency (some RDBMS drivers, HTTP based mechanisms). Some do not encourage interoperability and extensibility based on open IETF standards (RDBMS, Gnutella, Freenet).

**Chapter 8** summarizes the work presented in this thesis. We also outline interesting directions for future research.

Due to the different scope of each chapter, we split the comparison of our work with existing research results over the chapters 3, 4, 5, 6 and 7.

## 1.4 Terminology

The Internet Engineering Task Force (IETF) and World Wide Web Consortium (W3C) use well defined terms to help unambiguous interpretation of standards specifications. Accordingly, in specification sections, the keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in RFC-2119 [30]. This is particularly the case in chapters 4, 5, 6 and 7, even if these keywords are spelled in lower case.

# Chapter 2

# Service Discovery Processing Steps

## 2.1 Introduction

The most straightforward but also most inflexible configuration approach for invocation of remote services is to hard wire the location, interface, behavior and other properties of remote services into the local application. Loosely coupled decentralized systems call for solutions that are more flexible and can seamlessly adapt to changing conditions. The *web service* vision [6, 7] of distributed computing attempts to address the problem. Here programs are made more flexible and powerful by querying Internet databases (registries) at runtime in order to discover information and network attached third-party building blocks offering needed functionality under conditions matching a specification. For example, a data-intensive High Energy Physics analysis application looks for remote services that exhibit a suitable combination of characteristics, including network load, available disk quota, security options, access rights, and perhaps Quality of Service and monetary cost. A key question then is:

- *What distinct problem areas and processing steps can be distinguished in order to enable flexible remote invocation in the context of service discovery?*

This introductory chapter outlines eight problem areas and their associated processing steps, namely *description, presentation, publication, request, discovery, brokering, execution* and *control*. We propose a simple grammar (*SWSDL*) for describing network services as collections of service interfaces capable of executing operations over network protocols to endpoints. The grammar is intended to be used in the high-level architecture and design phase of a software project. A service must present its current description so that clients from anywhere can retrieve it at any time. For broad acceptance, adoption and easy integration of legacy services, an HTTP hyperlink is chosen as an identifier and retrieval mechanism (*service link)*. A registry for publication and query of service and resource presence information is outlined. Reliable, predictable and simple distributed registry state maintenance in the presence of service failure or misbehavior or change is addressed by a simple and effective soft state mechanism. The notions of *request, resource* and *operation* are clarified. We outline the discovery step, which finds services implementing the operations required by a request. The brokering step determines an invocation schedule, which is a mapping over time of unbound operations to service operation invocations using given resources. The execution step implements a schedule. It uses the supported protocols to invoke operations on remote services. We discuss how one can reliably support monitoring and controlling the lifecycle of

a request in the presence of a service that cannot reliably complete a request within a short and well-known expected timeframe.

## 2.2 Description

In a distributed system, it is desirable to describe (and maintain) the active participants such as services, resources and user communities, in order to allow for collaborative functionality. In particular, the description of services should encourage both flexibility and interoperability. As communications protocols and message formats are standardized on the Internet, it becomes increasingly possible and important to be able to describe communication mechanisms in some structured way. A service description language addresses this need by defining a grammar for describing network services as collections of service interfaces capable of executing operations over network protocols to endpoints. Service descriptions provide documentation for distributed systems and serve as a recipe for automating the details involved in application communication [8].

It is important to note that the concept of a service is a logical rather than a physical concept. The service interfaces of a service may, but need not, be deployed on the same host. They may be spread over multiple hosts across the LAN or WAN and even span administrative domains. This notion allows speaking in an abstract manner about a coherent interface bundle without regard to physical implementation or deployment decisions. We speak of a *distributed (local) service*, if we know and want to stress that service interfaces are indeed deployed across hosts (or on the same host). Typically, a service is persistent (long lived), but it may also be transient (short lived, temporarily instantiated for the request of a given user). Examples for a service are:

- A commodity FTP server.
- An FTP file server with an auxiliary registry interface for (secure) queries over access logs and statistics. The registry is deployed on a centralized high availability server that is remote to the core FTP server and shared by multiple such FTP servers of a computing cluster.
- An HTTP frontend to a database server, augmented with an auxiliary TCP buffer size tuning interface, as well as an event notification interface.
- A Java based HTTP process for job submission, augmented with an administration interface for shutdown and a PERL daemon periodically receiving and publishing current job status.

Description is the process of defining metadata for a thing that allows one to reason about the thing itself. In our context, sufficient service description metadata needs to be defined (and published) that allows a client to start communicating with the service. In support of this goal one describes in a structured manner ...

- Which interfaces a service offers

- Which operations and arguments are defined on an interface
- How operations and arguments are bound (mapped) to network protocols and endpoints

The formalism should be sufficiently general to encourage broad acceptance and smooth evolution. For example, it should not only support the needs of future services and protocols (e.g. using SOAP/HTTP [9] or SOAP/BEEP [31]), but should also be able to describe the typical operations of services using existing protocols such as FTP [32], SMTP [33], HTTP [34] and BEEP [35, 36].

Since the purpose of this section is to expose fundamental concepts rather than syntactic details, a precise formalism is not specified here. Instead, we now use examples to informally introduce a modified and strongly simplified form of the *Web Service Description Language (WSDL)* [8]. WSDL is a rigorous, expressive and flexible industry standard. It allows to take advantage of existing WSDL tools for automatic generation of client and server code from service descriptions. However, WSDL trades clarity for expressiveness and flexibility. The example stated in [8] requires 66 XML lines and 7 levels of XML [11] nesting even though it merely describes a stock quote service with a trivial operation that returns the trading price of a given stock. WSDL is ill suited for compact exposition of concepts and examples. Hence, the sole reason for introducing a new formalism here is that WSDL has the distinct disadvantage of being very complex and verbose. We call the new formalism *Simple WSDL (SWSDL)* and stress that it is a pedagogical vehicle, not an attempt to replace the standard[1]. We estimate that WSDL based service descriptions are about one order of magnitude larger in size and structural complexity than corresponding SWSDL based descriptions. All features of SWSDL can be (and in practice will be) mapped to WSDL. Both languages are not mutually exclusive. SWSDL is more useful in the high-level architecture and design phase of a software project whereas WSDL is more useful for the detailed specification and implementation phase.

SWSDL describes the interfaces of a distributed service object system. For simplicity, it offers neither a class concept nor interface inheritance. In SWSDL, a service description defines a *service* as a set of related *service interfaces*. A service interface has an *interface type*. An interface type defines a set of *operations* and *arguments*. The interface type can be used to check whether a service interface conforms to some well-known standard. An operation is bound to one or more protocols and network endpoints via *binding* definitions.

As an example, assume we have a simple scheduling service that offers an operation `submitJob` that takes a job description as argument. The function should be invoked via the HTTP protocol. A valid service description reads as follows:

```
<service>
  <interface type = "http://gridforum.org/interface/Scheduler-1.0">
    <operation>
      <name>void submitJob(String jobdescription)</name>
      <allow> http://cms.cern.ch/everybody </allow>
      <bind:http verb="GET" URL="https://sched.cern.ch/scheduler/submitjob"/>
    </operation>
  </interface>
</service>
```

[1]For example, the formalism does not clarify how input and output arguments of operations can be defined in a protocol independent type system, and how they can later be bound to a protocol dependent form.

The description above states that the service interface is of a scheduler type. The precise scheduler type, including syntax and semantics of operations, is identified by the URL `http://gridforum.org/interface/Scheduler-1.0`. Note that the URL is purely an identifier. It is left unspecified what (if anything) happens upon HTTP requests to the URL. For example, it is not required that a detailed technical paper can be found at the interface type URL. Next, we define `submitJob` as being the name of the submit operation and input and output arguments of type `String`. The operation is bound to the HTTP protocol, and can be invoked by sending an HTTP GET request to the URL `https://sched.cern.ch/scheduler/submitjob` (subject to local security policiy). Only members of the CMS virtual organization are allowed to invoke this operation. For security reasons, it is not specified who actually belongs to the CMS organization. The optional authorization hint is purely an identifier without explicitly specified semantics.

## 2.3 Presentation

Having outlined the structure of a service description, we now turn to the problem of description presentation. Clearly clients from anywhere must be able to retrieve the current description of a service (subject to security policy). Hence, a service needs to present (make available) to clients the means to retrieve the service description. To enable clients to query in a global context, some identifier for the service is needed. Further, a description retrieval mechanism is required to be associated with each such identifier. Together these are the bootstrap key (or handle) to all capabilities of a service. In principle, identifier and retrieval mechanisms could follow any reasonable convention.

In practice, however, a fundamental mechanism such as service discovery can only hope to enjoy broad acceptance, adoption and subsequent ubiquity if integration of legacy services is made easy. The introduction of service discovery as a new and additional auxiliary service capability should require as little change as possible to the large base of valuable existing legacy services, preferable no change at all. It should be possible to implement discovery-related functionality without changing the core service. Further, to help easy implementation the retrieval mechanism should have a very narrow interface and be as simple as possible.

The service description concept comes to our help. In support of these requirements, we logically separate core service functionality and presentation functionality into separate service interfaces. Further, the identifier is chosen to be a URL, and the retrieval mechanism is chosen to be HTTP(S). We define that an HTTP(S) GET request to the identifier must return the current service description (subject to local security policy). In other words, a simple hyperlink is employed. In the remainder of this thesis, we will use the term *service link* for such an HTTP URL identifier enabling service description retrieval.

Because service descriptions should describe the essentials of the service, it is recommended[2] that the service link concept be an integral part of the description itself. As a result, we extend the example description developed in the previous section with a *Presenter*

[2]In general, it is not mandatory for a service to implement any "standard" interface. Historical evidence suggests that the acceptance of ubiquituous Internet infrastructures and their flexible and successful evolution strongly depends on being conservative with the term *MUST*.

interface type. We propose that service descriptions can be retrieved via the Presenter, which defines an operation `getServiceDescription` for this purpose. The operation is identical to service description retrieval and is hence bound to (invoked via) an HTTP(S) GET request to a given service link. Additional protocol bindings may be defined as necessary. The new service description now reads:

```
<service>
  <interface type = "http://gridforum.org/interface/Presenter-1.0">
    <operation>
      <name>XML getServiceDescription()</name>
      <bind:http verb="GET" URL="https://sched.cern.ch/getServiceDescription"/>
    </operation>
  </interface>

  <interface type ="http://gridforum.org/interface/Scheduler-1.0">
    <operation>
      <name>void submitJob(String jobdescription)</name>
      <allow> http://cms.cern.ch/everybody </allow>
      <bind:http verb="GET" URL="https://sched.cern.ch/scheduler/submitjob"/>
    </operation>
  </interface>
</service>
```

## 2.4 Publication

Publication is the process of making the presence of services, resources, user communities and other metadata known to potential clients. In this specific context, it is the process of making a service identifier and description retrieval mechanism (in practice a service link) known to potentially interested clients so that they can retrieve the service description and use it. We are concerned with the basic capability of making a service link (and hence service description) *reachable* for clients. To this end, service links are collected in one or more well-known registries, which are databases that can be queried by clients. Registries for organizations or communities with special interests serve a similar purpose as link list web pages and top-level organizational web pages: Many clients use well known and authoritative websites (registries) as entry points for browsing and searching because they mostly contain relevant hyperlinks (service links) for the given target community. Consequently, a particular community can discover information relevant to its interests.

When a service starts up, it announces its presence by invoking a `publish` operation on the registry. The operation takes as argument a service link to identify the service attempting publication. The registry appends the service link if it is not already present. Conversely, when a service shuts down it announces its unavailability by invoking a `depublish` operation on the registry. The registry removes the service link if it is present. Clients can query the registry by invoking a query operation. The simplest query operation (`getLinks`) takes no arguments and returns the set of all known service links. An example result set for a query reads:

```
<tupleset>
```

```
    <tuple link="http://sched001.cern.ch/getServiceDescription"/>
    <tuple link="http://sched.infn.it:8080/pub/getServiceDescription"/>
    <tuple link="http://repcat.cern.ch/pub/getServiceDescription?id=4711"/>
</tupleset>
```

To retrieve the service descriptions of a result set, a client needs to establish a network connection for each service link in the result set. In principle, this is no problem, but in practice it can lead to prohibitive latency, in particular in the presence of large result sets. This is due to the very expensive nature of secure (and even insecure) connection setup. To address this problem, we define an additional query operation (`getTuples`) that returns service descriptions instead of associated service links. A registry implementation can use caching to reduce the number of connection setups and/or can use keep-alive connections to minimize setup time. As a consequence, this query operation may return outdated descriptions. More importantly, it may return only a partial result set; excluding any service descriptions the registry is not authorized to retrieve (service links are returned instead). This is the case if a security sensitive publisher refuses to delegate authorization to the registry, but only allows select clients to invoke its `getServiceDescription` operation. An example result set with two normal services and one replica catalog service refusing trust delegation may read:

```
<tupleset>
    <tuple link="http://sched001.cern.ch/getServiceDescription">
       <content>
         <service> service description A goes here </service>
       </content>
    </tuple>

    <tuple link="http://sched.infn.it:8080/pub/getServiceDescription">
       <content>
         <service> service description B goes here </service>
       </content>
    </tuple>

    <tuple link="http://repcat.cern.ch/pub/getServiceDescription?id=4711">
    </tuple>
</tupleset>
```

In this section, we deliberately describe only the simplest type of registry. This type roughly corresponds to the capabilities offered by the Universal Description, Discovery and Integration (UDDI) standard [10], with the exception that an UDDI registry offers slightly richer query capabilities and stores service descriptions rather than service links. Maintenance of service links and maintenance of service descriptions are two related but distinct concepts. Service link maintenance (SLM) is more fundamental than service description maintenance (SDM). We argue that a registry should primarily store dynamic pointers to external data, rather than data itself, for reasons of security, confidentiality, consistency, and perhaps efficiency:

- SDM can easily be layered on top of SLM.

- SLM allows keeping access control for security sensitive service descriptions in the hands of the authoritative Presenter. SDM, on the other hand, requires trust delegation, which potentially opens the door for a malicious audience to learn detailed enough service descriptions to launch well-focused virus or denial of service attacks. System administrators typically do not encourage unauthorized publication of information that exposes existing vulnerabilities or opens new security holes.

- Service links are smaller in size than service descriptions and can be published with less bandwidth consumption, or more often (see next section below).

- SLM reduces state consistency problems, since service links change much less frequently than the associated service descriptions.

Clearly many different types of sophisticated query capabilities can be introduced. The topic is discussed later in more depth. Here we only note that advanced query support can be expressed on top of the basic capabilities introduced so far, and that such higher-level capabilities conceptually do not belong to publication. Publication is only concerned with the fundamental capability of making a service link *reachable*[3] for clients. As an analogy, consider the related but distinct concepts of web hyper-linking and web searching: Web hyper-linking is a fundamental capability without which nothing else on the Web works. Many different kinds of web search engines using a variety of search interfaces and strategies can and are layered on top of web linking.

We will later also consider improvements over centralized registries by extending the discussion to fine-grained fully distributed registries.

## 2.5 Soft State Publication

This section discusses mechanisms for reliable, predictable and simple distributed registry state maintenance. In a system composed of a very large number of services, the mean time between failures is small. Recall that the previous section proposed a model in which services explicitly publish and de-publish as appropriate. We ignored the fact that services often fail or misbehave, leaving a registry in an inconsistent state. For example, a service that crashes may not de-publish with the registry, and hence clients may unnecessarily discover and try to contact an unavailable service repeatedly. Similarly, a service may be reconfigured to change its service link (and service description), yet it may forget to update all registries with which it is already associated. Further, a registry may change its authorization policy and, as a result, an already published service may suddenly no longer be in the position to de-publish itself. All these situations leave inconsistent or stale registry state behind. It is difficult for a registry to detect such situations and to determine when and how they can be resolved. For example, one can envision a strategy in which a registry drops a service link if a client (or perhaps itself) finds a service to be unavailable. However, the unavailability may be due to an authorization policy denying access to some but not all clients, or due to

[3] *Reachability* is interpreted in the spirit of garbage collection systems: A service link is reachable for a given client if there exists a direct or indirect retrieval path from the client to the service link.

problems in a small network segment or simply due to service reboot. The service owner may be offended and claim violation of a service level agreement (SLA), because, in his opinion, there is no reason for dropping its service. Even worse, the service owner might not even notice for quite some time that he has been dropped. To summarize, so-called *hard state* based distributed information systems populated from many independent autonomous and heterogeneous distributed sources typically evolve quickly into garbage dumps where valid information is hard to distinguish from trash, decreasing overall utility dramatically.

Elaborate mechanisms can be designed to cope with the problems of reliable and consistent state maintenance. Such mechanisms typically face many complex and subtle problems. However, one can elegantly avoid much complexity by using a simple *soft state* mechanism for reliable distributed garbage collection: State established at a remote location may eventually be discarded unless refreshed by a stream of subsequent confirmation notifications [37]. In this manner, component failures and changes are tolerated in the normal mode of operation rather than addressed through a separate recovery procedure [26]. Lack of refresh indicates service failure, shutdown or change.

The responsibility for state maintenance is displaced by moving it from the registry to the publishing services. Registries keep service links (and perhaps also descriptions) as soft state, that is, they are kept for a limited amount of time only. Service links are tagged with time-to-live tokens (TTLs). Service links are expired and dropped unless explicitly renewed via periodic publication, henceforth termed *refresh*. Services refresh by essentially saying, *"I am still here"*. Consequently, services can crash, stop, be added or changed without leaving stale state behind indefinitely.

For example, assume service descriptions change more frequently than service links. A job execution service publishes its current service link with a scheduler and asserts that the link is not expected to change within the next 2 hours, and that the current service description is not expected to change within the next 10 minutes. In other words, the service link or description may (but need not) be dropped if the execution service should not be able to refresh within the next 2 hours or 10 minutes, respectively. The execution service foresees temporary network failure, acts defensively and tries to confirm its presence more often, by refreshing its service link every 2 minutes. Even though the scheduler receives refreshes every 2 minutes, it need not retrieve and update the service description so often. It chooses to do so only every 10 minutes. In an effort to avoid dropping failed services that eventually manage to become available again, a (special-purpose) registry interface implementation moves expired service links from the main database into an auxiliary database, and later retries service description retrieval for two weeks, with exponentially growing delay between retries. Only after two weeks, it finally gives up and completely abandons the link.

Having discussed the motivation, context and scope of publication, a detailed design and specification can be developed. Chapter 4 is dedicated to this purpose.

## 2.6 Request

Let us clarify some terminology surrounding the formulation of requests from clients to use network attached third party functionality.

Clients formulate *requests*. Examples are: "submit job", "compute Pi with accuracy of 10000 decimals", and "retrieve result of HTTP GET to a given URL". Another example is an analysis job consisting of a sequence of operations. It first uses a file transfer service (to stage input data from remote sites), next a replica catalog service (to locate an input file replica with good data locality), then a job execution service (to run the analysis program), and finally again a file transfer service (to stage output data back to the user desktop).

*Resources* are things that can be used for a period of time, and may or may not be renewable. They have owners, who may charge others for using resources, and they can be shared or be exclusive. Examples include disk space, network bandwidth, specialized device time, and CPU time [38]. Resources are made accessible through the operations of services. *Operations* are consumers of resources.

*Requests* are hierarchical entities, and may have recursive structure; i.e., requests can be composed of sub requests or operations, and sub requests may themselves contain sub requests. The *leaves* of this structure are operations. The simplest form of a request is one containing a single operation. The definition is derived from [38]. Sometimes complex constraints and preferences formulated in a request description language accompany requests.

## 2.7 Discovery

Having formulated a request, the discovery step finds services implementing the operations required by a request. More precisely, for each operation of a request of a given user, the discovery step searches one or more registries and produces *candidate services*, which are services (more precisely: service descriptions) that implement the operation on top of a given set of protocols. The simplest form of candidate contains a single service implementing a single operation on top of a single protocol.

It is often necessary to use several services in combination to implement the operations of a request. For example, a request may involve the combined use of a file transfer service (to stage input and output data from remote sites), a replica catalog service (to locate an input file replica with good data locality), a request execution service (to run the analysis program), and finally again a file transfer service (to stage output data back to the user desktop). Hence, discovery often involves querying for several types of operations or services.

## 2.8 Brokering

For each operation of a request of a given user, the previous discovery step produces a set of candidate services that implement the operation. In the following brokering step, more or less sophisticated techniques are used to refine the selection and determine an invocation schedule. *Schedules* (also termed *execution plans*) are mappings over time of unbound operations to service operation invocations using given resources. One maps operations, not requests, because requests are containers for operations, and operations are the actual resource consumers [38]. The simplest schedule contains a single service operation.

The brokering step can be as simple as randomly picking a single service from the candidates, or as sophisticated as initiating a complex auction where participating services place

bids and negotiate a resolution based on economic models, Quality of Service (QoS) and/or Service Level Agreements (SLAs). Consider a less ambitious example where, in an attempt to minimize response time, the brokering step of a job scheduler compares the current CPU load of candidate job execution services.

As mentioned above, it is often necessary to use several services in combination to implement the operations of a request. In such cases it is often helpful to consider correlations. For example, a scheduler for data-intensive requests may look for input file replica locations with a fast network path to the execution service where the request would consume the input data. If a request involves reading large amounts of input data, it may be a poor choice to use a host for execution that has poor data locality with respect to an input data source, even if it is very lightly loaded.

An advanced job scheduler typically also matches execution services against query patterns describing job requirements such as desired operating system and computer architecture type, minimum main memory size, disk quota, availability and connectivity to third party services like database engines, etc. Requests with complex constraints and preferences, for tasks like job submission, are augmented with a structured request description language for matching and ranking.

As can be seen, advanced brokering for tasks like job scheduling often requires additional information not available as part of service descriptions. Such additional information must be gained from other data sources. Such data sources for brokering may follow and respect a single globally standardized data and query model. In practice, however, non-uniform special-purpose data sources are often involved. This is due to the heterogeneous nature of large distributed cross-organizational systems such as the Grid, the large variety of use cases, brokering strategies and query types as well as strongly varying data freshness and data aggregation requirements. For example, brokering information includes very slowly changing data, such as the type of operating system, or more frequently changing quantities, such as the number of running jobs or the current CPU utilization. In addition, the brokering process may be highly application specific and due to its complexity not expressible in any known query language. For example, it is hard to envisage that a negotiation process involved in distributed auctions can be expressed as a query (rather than an algorithm). In special-purpose areas, special matchmaking mechanisms have been developed [39].

## 2.9 Execution

In the previous brokering step, more or less sophisticated techniques are used to determine an invocation schedule from candidate services. The execution step implements a schedule. The service descriptions of the operations of a schedule are parsed, and the supported protocols are used to invoke operations on remote services. In an attempt to cover a broad range of existing and future protocols, *invocation* is understood very broadly. For example, the operation to be invoked may be a SOAP operation carried over BEEP or HTTP(S), but it may also simply mean issuing one of the standard commands supported by the FTP or SMTP protocol (e.g. `DELETE, GET`).

In some cases more or less advanced resource reservations on a set of services can ac-

company the execution step in order to help ensure consistent execution semantics for the operations of a request, or to guarantee a certain Quality of Service (QoS). For a detailed discussion of Quality of Service and advanced reservation see [40]. Note that services controlling a domain can commit resources with authority, whereas services outside a control domain cannot do so. For example, a local scheduler managing a cluster is in the position to make definitive statements about its resource usage and policy in general. A cross-organizational global scheduler, on the other hand, does not own any cluster, and hence cannot commit resources with authority. Such a global scheduler can only compute schedules based on assumptions and educated guesses.

## 2.10 Control

It is critical for practical handling of resource-intensive jobs that a disconnected mode of operation is supported, as well as monitoring and tracking of request progress and controlling the lifecycle of ongoing requests. Examples requiring such capabilities include a simulation requiring a week of CPU time, transfer of a large set of files via WAN to a robotic tape library, and an analysis sweeping over Gigabytes to Terabytes of data.

A service can implement an operation with a synchronous and/or asynchronous model of invocation. *Synchronous invocation* is a call-and-wait-until-done model. The client sends a request and blocks until the service has completed the request and the client has received the response (or an error code). Due to its simplicity, this model is very popular. For example, the vast majority of programming library functions and many web transactions follow it. It is most appropriate in situations where a service can be expected to reliably complete a request within a short and well-known timeframe. There are three types of problems related to the synchronous model.

- **Cannot monitor, control and maintain life cycle.** If for some reason, no response is received within an expected timeframe, a client usually starts wondering whether there is a problem, what it may be and what could possibly be done to resolve it. A client may want to ask for priority change, suspension, resumption, rescheduling or other such lifecycle maintenance. However, a client cannot refer to the original request without having a unique request identifier, and hence cannot establish monitor and control connections. Consequently, there is no way for a client to find out whether the request has been aborted incorrectly, whether it is waiting on some locked resource, or whether it is fine and just about to be completed. Consider services accepting heavy-duty requests or opaque and potentially unbounded requests (e.g. submission of arbitrary executables or database queries) from a large and diverse user community. Such services are usually fully loaded, and typically cannot give meaningful timeframes for reliable request completion.

- **Cannot work in a disconnected mode.** Assume that on a Friday a user submits a request that takes the weekend for completion. The user may want to logout or shutdown his laptop without losing the request and come back on Monday to retrieve results. A client shutdown may also occur due to some unexpected problem such as an

operating system crash. With synchronous invocation, a network connection between client and service must be kept alive for the entire request lifecycle. If the network connection dies for some reason, the service aborts the request. It would be useless for the service to continue processing the request because a client would have no way to retrieve results through subsequent control connections, as the client cannot refer to the original request without having a unique request identifier. Synchronous invocation is not designed to work in a disconnected mode.

- **Early exhaustion of system resources.** In practice, services allocate a constant amount of system resources per request just to be able to start handling it. Examples include a TCP connection on a port, a thread or process, and some memory. Only a finite amount of these resources is available from the operating system, and they usually cannot be reclaimed before request completion. Therefore, in practice there exists an upper bound on the number of requests a service can handle concurrently. This is the case even if most requests are waiting in a persistent queue and not running. This upper bound is often very low (e.g. 50-1000 requests), severely limiting scalability and leading to surprisingly early service denial errors.

A richer invocation model is needed to address these shortcomings, which we now discuss.

*Asynchronous (non-blocking) invocation* implements a more powerful, but also more complex invocation model. The important aspect of this model is that it allows for monitoring and tracking of request progress as well as controlling the lifecycle of ongoing requests. It also allows for disconnected operation and low resource consumption. Bear in mind, however, that because of its complexity the asynchronous invocation model is considered overkill for many if not most services (e.g. replica catalog lookup service, time service, logging service, security credential storage service). Its usage in current systems is the exception rather than the rule. With asynchronous invocation, a client sends a request; the service accepts the request and immediately returns a unique request handle as an identifier, even though the request has not yet completed. The client continues to do other useful work while the service is busy handling the request. There is now a choice of pull or push based implementation. In the pull-based case, the active client periodically polls the passive service, checking the status of the request identified with the given handle (e.g. queued, suspended, resumed, aborted, percent completed). In order to enable a client to react without a need to poll repeatedly, a push-based variant can also be used. In such a case, an active service notifies the passive client of registered events or state transitions in the request life cycle. An asynchronous service is well conditioned to handle large numbers of concurrent requests, because the resources required per request can be kept minimal. A request moved into a persistent wait queue requires no TCP connection, no thread and almost no memory. Although there exist variations to the pull/push theme, they do not differ enough to warrant further exposition in this context.

In any case, the main point is that a client can disconnect and is in the position to monitor the status of a request, abort it if so desired, or perhaps even ask for priority change, suspension, resumption, rescheduling or other such lifecycle maintenance.

## 2.11 Summary

This introductory chapter outlines eight problem areas of service and resource discovery, and their associated processing steps, namely *description, presentation, publication, request, discovery, brokering, execution* and *control*.

**Description.** A service description language encourages flexibility and interoperability by defining a grammar for describing network services as collections of service interfaces capable of executing operations over network protocols to endpoints. Service descriptions provide documentation for distributed systems and serve as a recipe for automating the details involved in application communication. In the pedagogical SWSDL language, a service description defines a *service* as a set of related *service interfaces*. A service interface has an *interface type*. An interface type defines a set of *operations* and *arguments*. The interface type can be used to check whether a service interface conforms to some well-known standard. An operation is bound to one or more protocols and network endpoints via *binding* definitions.

**Presentation.** A service must present its current description so that clients from anywhere can retrieve it at any time. An identifier and a description retrieval mechanism are the key to all capabilities of a service. For broad acceptance, adoption and easy integration of legacy services, an HTTP hyperlink is chosen as an identifier and retrieval mechanism (*service link)*. It is recommended that the service link is made an integral part of the description itself.

**Publication.** Publication is the process of making the presence of services, resources, user communities and other metadata reachable to potential clients. To this end, service links are collected in one or more well-known database registries. When a service starts up, it announces its presence by invoking a *publish* operation on the registry. Clients can query the registry by invoking a *query* operation.

**Request.** Clients formulate requests. Resources have owners, who may charge others for using resources. Examples include disk space, network bandwidth and CPU time. Resources are made accessible through the operations of services. Operations are consumers of resources. Requests are composed of operations.

**Discovery.** The discovery step finds services implementing the operations required by a request. More precisely, for each operation of a request of a given user, the discovery step searches one or more registries and produces *candidate services*, which are services that implement the operation on top of a given set of protocols.

**Brokering.** In the brokering step, more or less sophisticated techniques are used to refine the selection and determine an invocation schedule. *Schedules* (also termed *execution plans*) are mappings over time of unbound operations to service operation invocations using given resources. The brokering step can be as simple as randomly picking a single service from the candidates, or as sophisticated as initiating a complex auction. It is often necessary to use

several services in combination to implement the operations of a request, in which case it is helpful to consider correlations. Sometimes requests are augmented with complex constraints and preferences for matching and ranking.

**Execution.** The execution step implements a schedule. The service descriptions of the operations of a schedule are parsed, and the supported protocols are used to invoke operations on remote services.

**Control.** A service can implement an operation with a synchronous and/or asynchronous model of invocation. *Synchronous invocation* is a call-and-wait-until-done model. The client sends a request and blocks until the service has completed the request and the client has received the response. It is most appropriate in situations where a service can be expected to reliably complete a request within a short and well-known timeframe. The model has three problems: It cannot monitor, control and maintain the life cycle of ongoing requests. It cannot work in a disconnected mode. Further, it may lead to early exhaustion of system resources. *Asynchronous (non-blocking) invocation* addresses these shortcomings. A client sends a request; the service accepts the request and immediately returns a unique request handle as an identifier, even though the request has not yet completed. The client continues to do other useful work while the service is busy handling the request. Then (in pull-mode), the active client periodically polls the passive service, checking the status of the request identified with the given handle. A push-based variant can also be used.

# Chapter 3

# A Data Model and Query Language for Discovery

## 3.1 Introduction

In a distributed system, it is desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. As in a data integration system, the goal is to exploit several independent information sources as if they were a single source. This enables information discovery and collective collaborative functionality that operates on the system as a whole, rather than on a given part of it. In a large distributed system spanning many administrative domains, the set of information tuples in the universe is partitioned over one or more nodes from a wide range of distributed system topologies, for reasons including autonomy, scalability, availability, performance and security. A database model is required that clarifies the relationship of the entities in a distributed system.

The distribution and location of tuples should be transparent to a query. However, in practice, it is often sufficient (and much more efficient) for a query to consider only a subset of all tuples (service descriptions) from a subset of nodes. For example, a typical query may only want to search tuples (services) within the scope of the domain `cern.ch` and ignore the rest of the world. Both requirements need to be addressed by an appropriate query model.

A data model remains to be specified. It should specify what kind of data a query takes as input and produces as output. Due to the heterogeneity of large distributed systems spanning many administrative domains, the data model should be flexible in representing many different kinds of information from diverse sources, including structured and semi-structured data. The key problem then is:

- *In a large heterogeneous distributed system spanning many administrative domains, what kind of database, query and data model as well as query language can support simple and complex dynamic information discovery with as few as possible architectural and design assumptions? How can one uniformly support queries in a wide range of distributed system topologies and deployment models, while at the same time accounting for their respective characteristics?*

This chapter develops a database and query model as well as a generic and dynamic data model that address the given problem. A distributed database framework is used where there

exist one or more nodes that are interconnected with links, each node holding a database. Unlike in the relational model the elements of a tuple in our data model can hold structured or semi-structured data in the form of any arbitrary well-formed XML [11] document or fragment. An individual tuple element may, but need not, have a schema (XML Schema [12]), in which case the element must be valid according to the schema. The elements of all tuples may, but need not, share a common schema. A tuple is a multi-purpose data container that may contain arbitrary content. The concepts of (logical) query and (physical) query scope are cleanly separated rather than interwoven. A query is formulated against a global database view and is insensitive to link topology and deployment model. In other words, to a query the set of all tuples appears as a single homogenous database, even though the set may be (recursively) partitioned across many nodes and databases. The query scope, on the other hand, is used to navigate and prune the link topology and filter on attributes of the deployment model. A query is evaluated against a set of tuples. The set, in turn, is specified by the scope. Conceptually, the scope is the input fed to the query. Example service discovery queries are given. Three query types are identified, namely *simple, medium* and *complex*. An appropriate query language (XQuery [18]) is suggested. The suitability of the query language is demonstrated by formulating the example prose queries in the language. Detailed requirements for a query language supporting service and resource discovery are given. The capabilities of various query languages are compared.

## 3.2 Database and Query Model

**Database Model.** A distributed database framework is used where there exist one or more nodes. Each node can operate autonomously. A node holds a set of tuples in its database. A given database belongs to a single node. For flexibility, the databases of nodes may be deployed in any arbitrary way (*deployment model*). For example, a number of nodes may reside on the same host. A node's database may be co-located with the node. However, the databases of all nodes may just as well be stored next to each other on a single central data server. The database tuples may be dynamically (re) computed on each query. A database may be anything that accepts queries from the query model and returns results according to the data model (see below).

The set of tuples in the universe is partitioned over the nodes, for reasons including autonomy, scalability, availability, performance and security. Nodes are interconnected with links in any arbitrary way. A link enables a node to query another node. A *link topology* describes the link structure among nodes. The centralized model has a single node only. For example, in a service discovery system, a link topology could tie together a distributed set of administrative domains, each hosting a registry node holding descriptions of services local to the domain. Figure 3.1 depicts three example link topologies, namely ring, tree and graph. Chapter 7 outlines many more useful link topologies. Depending on the application context, all topologies have their merits and drawbacks.

**Query Model.** Our query model is intended for read-only search. Insert, update and delete capabilities are not required and not addressed. It is a general-purpose query model

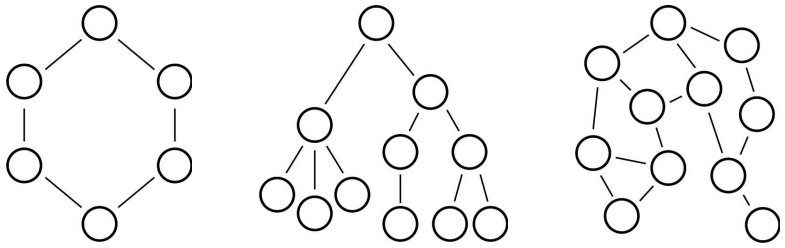

Figure 3.1: Ring, Tree and Graph Topology [41].

that operates on *tuples*. Discussion in this chapter often uses examples where the term *tuple* is substituted by the more concrete term *service description*.

In practice, it is often sufficient (and much more efficient) for a query to consider only a subset of all tuples (service descriptions) from a subset of nodes. For example, a typical query may only want to search tuples (services) within the scope of the domain `cern.ch` and ignore the rest of the world.

However, it is a strong user requirement that queries should be as insensitive as possible to any link topology and deployment model. In other words, a user should not need to reformulate a query when the node topology or deployment model changes, as is frequently the case in large distributed systems spanning many administrative domains such as P2P networks or cross-organizational Grids. A query model should not make any assumptions on the underlying database and query processing technology. P2P query engines, distributed database systems and centralized database systems should be able to answer the same queries. As in a data integration system [42, 43, 44], the goal is to exploit several independent information sources as if they were a single source. In support of these requirements, the concepts of (logical) query and (physical) query scope are cleanly separated rather than interwoven.

- **Query.** A query is formulated against a global database view and is insensitive to link topology and deployment model. In other words, to a query the set of all tuples appears as a single homogenous database, even though the set may be (recursively) partitioned across many nodes and databases. This means that in a relational or XML environment, at the global level, the set of all tuples appears as a single, very large, table or XML document, respectively.

- **Query Scope.** The query scope, on the other hand, is used to navigate and prune the link topology and filter on attributes of the deployment model. Searching is primarily guided by the query. Scope hints are used only as necessary. A query is evaluated against a set of tuples. The set, in turn, is specified by the scope. Conceptually, the scope is the input fed to the query. The query scope is a set and may contain anything from all tuples in the universe to none.

A query scope is specified either *directly* or *indirectly*. For example, one can directly enumerate the tuples (service descriptions) to be considered. However, this is usually impractical. One can also indirectly define a query scope by specifying a set of nodes (or Internet domain

names or table names), implying that the query should be evaluated against the union of all tuples contained in their respective databases. This corresponds to the concept of horizontally partitioned tables extensively used in large-scale relational database systems, in particular for distributed instances [45]. One can also indirectly specify the query scope by giving a time deadline, implying that as many tuples as possible should be considered, but only until the deadline has passed. Many more ways to specify a query scope can be envisioned. Both query and scope can prune the search space, but they do so in a very different manner. More detailed discussion is deferred to Section 6.8.

## 3.3 Generic and Dynamic Data Model

**Generic Data Model.** In a large distributed system, a registry is populated from a large variety of heterogeneous remote data sources. The input and output of a query are instances of a generic data model, which in our case is XML based and models a document as a tree of nodes. XML [11] is used because one of its strengths is its flexibility in representing many different kinds of information from diverse sources, including structured and semi-structured data. XML is, above all else, a unifying integration technology.

The data model must be capable of modeling an XML document as well as a well-formed fragment of a document, a sequence of documents, or a sequence of document fragments. We note that there is no need to store tuples in XML; they just need to be presented this way, perhaps by middleware. For example, it is common to present data from relational databases, dynamic content generation systems and legacy command line tools as XML. A more sophisticated system can accept queries over an XML view and internally translate the query into SQL [46, 47, 48, 43].

The data model represents a set of tuples. A *tuple* has, unsurprisingly, zero or more XML attributes and zero or more XML elements. An example tuple reads as follows:

```
<tuple id="123">
  <patient>
     <name> "Doe,John" </name>
     <address> "1, South St., Palm Beach" </address>
  </patient>
  <record>
    <entry date="1/9/90" diagnosis="amnesia"/>
  </record>
</tuple>
```

In the relational model, a tuple has a number of column values. All tuples of all nodes are homogenous in the sense that their column values comply with a single strongly typed schema. In our model, this is not required. The elements (columns) of a tuple can hold structured or semi-structured data in the form of any arbitrary well-formed XML document or fragment. An individual tuple element may, but need not, have a schema, in which case the element must be valid according to the schema. The elements of all tuples may, but need not, share a common schema. An element (column) is typed (`type XML`), but obviously in a very loose manner. A tuple is a multi-purpose data container that may contain arbitrary content. Unlike in a RDBMS, a single (logical) tuple set contains *all* tuples. This implies that a query

need not specify a "table" or "tuple set name" to indicate the type of tuples that should be considered. Rather, predicates within the regular query language are used to select the desired tuples from the single set. Arguably, it is more appropriate to adopt XML parlance and also use the term *element* instead of *tuple*. Nevertheless, continuing to use established terminology from the relational world seems to improve clarity more than it is misleading. Discussion in this chapter often uses examples where the term *tuple* is substituted by the more concrete term *service description*.

The actual query is fed as input an XML representation that has the following form.

```
<tupleset>
   zero or more tuples go here
</tupleset>
```

The output of a predicate query (see below) is a subset of its input. The output of a constructive query (see below) is an arbitrary structure of the following form.

```
<tupleset>
   zero or more XML elements go here
</tupleset>
```

In any case, the query engine always encapsulates the query output with a `tupleset` root element. A user query need not generate this root element as it is implicitly added by the environment.

**Dynamic Data Model (DDM).** In a large distributed system spanning many administrative domains, a registry is populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources. To address dynamic state maintenance problems, we propose a *Dynamic Data Model (DDM)*. In DDM, a tuple is an annotated multi-purpose soft state data container that may contain a piece of arbitrary *content*. Examples for content include a service description, file, picture, current network load, host information, stock quotes, etc. Content of a given *type* is maintained for a given *context* (purpose) and may be associated with some *metadata*. A tuple and its content are valid for some time span only. At any time, the current (up-to-date) content can be retrieved from the authoritative content provider via a dynamic pointer called a *content link*. The pointer can be used if stale content is to be avoided.

The Dynamic Data Model is an instantiation of the Generic Data Model where a tuple has as attributes a content link, a context, a type, and some time stamps. Optionally, each tuple also has a metadata element and content extensibility element. Detailed justification is deferred to Section 4.2. Consider the following example dynamic tuple.

| **Link** | **Context** | **Type** | **TS1** | **TS2** | **TS3** | **Metadata** | **Content** |
|---|---|---|---|---|---|---|---|
| http://sched001.cern.ch/ getServiceDescription | Parent | Service | 10 | 20 | 30 | <owner name = "http://cms.cern.ch"/> | <service> A < /service> |

The actual query is fed as input an XML representation that has the following form (discussion of time stamps `TS` is deferred).

```
<tupleset TS4="100">
    <tuple link="http://sched001.cern.ch/getServiceDescription"
           type="service" ctx="parent" TS1="10" TC="15" TS2="20" TS3="30">
       <content>
         <service> service description A goes here </service>
       </content>
       <metadata>
         <owner name="http://cms.cern.ch"/>
       </metadata>
    </tuple>

    <tuple link="http://repcat.cern.ch/pub/getServiceDescription?id=4711"
           type="service" ctx="child" TS1="30" TC="0" TS2="40" TS3="50">
    </tuple>

    <tuple link="http://monitor.cern.ch/pub/getHostInfo"
           type="hostInfo" TC="65" TS1="60" TS2="70" TS3="80">
       <content>
         <hostInfo>
             <host name="fred01.cern.ch" os="redhat 7.2" arch="i386"
                   mem="512M" MHz="1000" cpus="2"/>
             <host name="fred02.cern.ch" os="solaris 2.7" arch="sparc"
                   mem="8192M" MHz="400" cpus="64"/>
         </hostInfo>
       </content>
    </tuple>
</tupleset>
```

## 3.4 Query Examples and Types

To concretize discussion and to identify query types and requirements, we now give several example queries related to service discovery. Queries are initially expressed in prose. They will later be formalized in a suitable query language. One can distinguish three types of queries: *simple, medium* and *complex*. The latter are more powerful than the former. Nevertheless, we will see that even a simple query is a powerful tool.

Section 6.5 will later show that different execution strategies are required to answer different query types. In short, complex queries are not recursively partitionable and hence are inefficient to answer without centralized database architectures. On the other hand, simple and medium queries are recursively partitionable, and hence can also effectively be answered in fully decentralized environments where tuples are physically partitioned among many small and independent nodes.

**Simple Query.** Simple queries are most often used for discovery. A simple query finds all tuples (services) matching a given predicate or pattern. The query visits each tuple (service description) in a set individually, and generates a result set by applying a function to each tuple. The function usually consists of a predicate and/or a transformation. Individual answers are added to a (initially empty) result set. An empty answer leaves the result set unchanged. A simple query has the following form:

```
R = {}
for each tuple in input
  R = R UNION { function(tuple) }
endfor
return R
```

Example simple queries are:

- *(QS1) Find all (available) services.*
- *(QS2) Find all services that implement a replica catalog service interface and that CMS members are allowed to use, and that have an HTTP binding for the replica catalog operation "XML getPFNs(String LFN)".*
- *(QS4) Find all local services (all service interfaces of any given service must reside on the same host).*
- *(QS5) Find all services and return their service links (instead of descriptions).*
- *(QS6) Find all CMS replica catalogs and return their physical file names (PFNs) for a given logical file name (LFN); suppress PFNs not starting with "ftp://".*
- *(QS7) Within the domain "cern.ch", find all execution services and their CPU load where cpuLoad $< 0.5$ (Assuming the operation cpuLoad() is defined on the execution service).*

Some typical simple queries posed to Peer-to-Peer file sharing networks are:

- *(Q20) Find all MP3 music files titled "Like a virgin".*
- *(Q21) Find all URLs under which replicas of a given file with identifier X (e.g. logical file name) can be downloaded.*
- *(Q22) Find all MP3 music files produced by artist Madonna since 1995, where download speed is above 100 KB/s.*

In support of the wide variety of real-life questions anticipated, it should be possible to arbitrarily combine and nest all capabilities exposed in these examples. Note that the first four queries return service descriptions, whereas the others return additional or entirely different information (service links, physical file names or CPU load). We term the former queries *predicate (or filter) queries*. The structure of the result set is predetermined in the sense that query output must be a subset of query input. We term the latter queries *constructive queries*, because they construct answers of arbitrary structure and content. Predicate queries are a subset of constructive queries. A constructive query function that always returns `"Hello World"` or an empty string is legal, but not very useful.

Further, note that the queries *QS6, QS7, QS22* involve multiple independent data sources and match on dynamically delivered content (via remote invocation of operations `getPFNs`

and `cpuLoad`), rather than on values being part of service descriptions. We call these queries *dynamic queries*, as opposed to *static queries*. To support dynamic queries, a query language must provide means to dynamically retrieve and interpret information from diverse remote or local sources.

Dynamic queries can sometimes be reformulated as static queries. For example, the LFN/PFN database information of query *QS6* could be made available as part of the tuple set. In practice, this is typically infeasible for reasons including database size, consistency, information hiding, security and performance. Publishing highly volatile attributes such as CPU load as part of tuples leads to stale data problems. Clearly dynamic invocation is a more appropriate vehicle to deliver CPU load. Alternatively, custom push protocols can be used, for example as defined in the Grid Monitoring Architecture [49].

**Medium query.** A medium query computes an answer over a set of tuples (service descriptions) as a whole. For example, it can compute aggregates like number of tuples, maximum, etc. Examples of medium queries are:

- *(QM1) Find the CMS storage service with the largest network bandwidth to my host "dummy.cern.ch" (assuming there exists a service estimating bandwidth from A to B).*
- *(QM2) Return the number of replica catalog services.*
- *(QM3) Find the two CMS execution services with minimum and maximum CPU load and return their service descriptions and load.*
- *(QM4) Return the services owned by members of the black sheep list (assuming the tuple set not only contains service descriptions, but also a set of black sheep tuples).*
- *(QM5) Return a summary of all replica catalogs and schedulers residing within the domains "cern.ch", "infn.it" and "anl.gov", grouped in ascending order by owner, domain and service type, with aggregate group cardinalities. A sample result set should look as follows:*

```
<tupleset>
  <owner name="alice.org" domainCount="2">
    <domain name="cern.ch" typeCount="2">
      <type name="http://gridforum.org/interface/ReplicaCatalog-1.0"
          serviceCount="13" />
      <type name="http://gridforum.org/interface/Scheduler-1.0"
          serviceCount="4" />
    </domain>
    <domain name="infn.it" typeCount="1">
        ...
    </domain>
  </owner>
  <owner name="cms.org" domainCount="3">
      ...
  </owner>
</tupleset>
```

Some typical medium queries posed to Peer-to-Peer file sharing networks are:

- *(Q25) Find the number of MP3 music files titled "Like a virgin" that have a size < 5 MB.*

- *(Q26) Find the top three MP3 music files titled "Like a virgin" with maximum bandwidth connectivity to my host "dummy.cern.ch" and return (URL, bandwidth) pairs.*

The query is applied to the set as a whole. For example, *QM4* is interesting in that it involves crossing tuple boundaries, which simple hierarchical query languages typically do not support. Like a simple query, a medium query can be static or dynamic. It can be a predicate query or a constructive query.

**Complex query.** Complex queries are most often used for advanced discovery or brokering. Like a medium query, a complex query computes an answer over a set of tuples (service descriptions) as a whole. However, it has powerful capabilities to combine data from multiple sources. For example, it supports all database join flavors. Like any other query, a complex query can be static or dynamic. It can be a predicate query or a constructive query. Example complex queries are:

- *(QC1) Find all (execution service, storage service) pairs where both services of a pair live within the same domain. (Job wants to read and write locally).*

- *(QC2) Find all hosts that run more than one replica catalog with CMS as owner. (Want to check for anomalies).*

- *(QC3) Find the top 10 owners of replica catalog services within the domains "cern.ch", "infn.it" and "anl.gov", and return their email, together with the number of services each of them owns, sorted by that number.*

## 3.5 XQuery Language

XQuery [18, 50, 51, 52] is the standard XML [11] query language developed under the auspices of the W3C. An understanding of the language is essential to the discussion in the remainder of this thesis. However, a number of excellent introductions and compact summaries of its features have already appeared. It is not considered valuable to generate even more. Therefore, we repeat select portions of the XQuery specification [18] in the following subsection, and of work by Manulescu, Florescu and Kossmann [48] in subsection "Language Features and Examples". Afterwards, the suitability of the language for service and resource discovery is demonstrated by translating example simple, medium and complex prose queries to the language.

### Introduction [18]

With the emergence of XML, the distinctions among various forms of information, such as documents and databases, are quickly disappearing. XML is an extremely versatile markup language, capable of labeling the information content of diverse data sources including structured and semi-structured documents, relational databases, and object repositories. A query language that uses the structure of XML intelligently can express queries across all these kinds of data, whether physically stored in XML or viewed as XML via middleware.

XQuery is designed to be a small, easily implementable language in which queries are concise and easily understood. The language is derived from an XML query language called Quilt [53], which in turn borrowed features from several other languages. From XPath [54] and XQL [55] it took a path expression syntax suitable for hierarchical documents. From XML-QL [56] it took the notion of binding variables and then using the bound variables to create new structures. From SQL [19] it took the idea of a series of clauses based on keywords that provide a pattern for restructuring data (the SELECT-FROM-WHERE pattern in SQL). From OQL [57] it took the notion of a functional language composed of several different kinds of expressions that can be nested with full generality.

XQuery is a functional language in which a query is represented as an expression. The principal forms of expressions are as follows: XPath expressions, element constructors, FLWR expressions, expressions involving operators and functions, conditional expressions, quantified expressions and expressions that test or modify data types.

The input and output of a query are instances of a data model, which is also used by XPath 2.0 [58]. A document is modeled as a tree of nodes. The data model is capable of modeling not only an XML document but also a well-formed fragment of a document, a sequence of documents, or a sequence of document fragments. XPath is a W3C standard notation for navigating along "paths" in an XML document, and is used in several XML-related standards including XSLT [59] and XPointer [60]. The type system follows XML Schema [12].

### Language Features and Examples [48]

**Path Expressions.** XQuery uses XPath [54] expressions for addressing parts of an XML document. The `document("http://mysite.org/med.xml")` expression retrieves the root of the XML document situated at the given URL. `document("med.xml")/record` is the ordered list of all `record` children of the document root; `document("med.xml")//record` returns the list of `record` elements at any depth in the document, in document order. Within a list, an indexing operator can be used: `document("med.xml")//record[1]` selects the first `record` in the document. A path may refer to a well-defined document, as in the previous examples, or is interpreted with respect to the root of a current document that is deduced from the evaluation context; the expression `//entry/@ssNo` retrieves the collection of values of the `ssNo` attributes in all `entry` elements in the current document. A dereference operator is also provided: `//entry/@rel_previous->entry` returns all medical entries that are "pointed at" by some other entry in the current document. Path predicates, interspersed in path expressions, restrict the navigation; for example, `//entry[date="1/9/90"]` returns all the

`entry` elements having at least one date child, whose string values is `"1/9/90"`. A path predicate can also select a specific range, e.g. `document("med.xml")//entry[range 2 to 5]` will return the second to fifth `entry` elements, in document order.

```
FILE med.xml
------------
<medical>
  <patient SSno="123">
     <name> "Doe,John" </name> <dob> "1/1/1960" </dob>
     <address> "1, South St., Palm Beach, FL" </address>
  </patient>
  <patient SSno="101">
     <name> "Ale, Mary" </name> <dob> "2/6/1970" </dob>
     <address> "2, Pine Rd., Bear Canyon, MN" </address>
  </patient>

  <record> <patientSSno> "123" </patientSSno>
    <entry entID="1">
       <date>"1/9/90"</> <symptoms>"fatigue, bad sleep"</>
       <diagnosis> </> <medication> "blood tests" </>
    </entry>
    <entry entID="2" rel_previous="1">
       <date>"10/9/90"</> <symptoms>"low blood iron"</>
       <diagnosis> "Anemy" </> <medication> "Biofer once a day" </>
    </entry>
  </record>
</medical>
```

**Operators.** XQuery provides the usual set of first-order operators (arithmetic, logical and set-oriented); the comma is a list concatenation operator. For example, `(//entry, //name)` returns the concatenation of the list of all entries, followed by all names, in document order. Second order operators in XQuery are the logical quantifiers `ANY`, `ALL`, and `SORT`. For example, `document("med.xml")//entry SORTY BY date DESC` will return all entry elements, the most recent first.

**Element Constructors.** These expressions provide for construction of new XML structures; for example, `<alphalist> document("med.xml")//name </alphalist>` constructs an alphabetic list of all patient names. Syntactically, the element's tag, attribute names and attribute values can be gained from constants or variables, while the children are specified as a list of arbitrary expressions.

**FLWR Expressions.** FLWR expressions (pronounced *"flower"*) consist of three parts: a `FOR-LET` clause that makes variables iterate over the result of an expression or binds variables to arbitrary expressions, a `WHERE` clause that allows specifying restrictions on the variables, and a `RETURN` clause that can construct new XML elements as output of the query. For example, the following query retrieves all the medical records of people with health problems that have been related to pollution within the last ten years.

```
FOR $r in document("med.xml")//record,
    $e in $r/entry
WHERE $e/date > "1/1/90" and contains($e/diagnosis, "pollution")
RETURN <pollutionIncident> $r/@ssNo, $e/diagnosis </pollutionIncident>
```

Variables defined with a `FOR` are iterators: `$r` variable iterates over all the record elements, and `$e` over the entries in the record associated to `$r`. For each `($r, $e)` pair that satisfies the condition, a new `pollutionIncident` element is created, containing the patient `SSno` and the diagnosis. The following variant `groups` the interesting entries according to the patient's `SSno`.

```
FOR $no in distinct(document("med.xml")//record/@ssNo)
LET $recs := document("med.xml")//record[@ssNo=$no]
RETURN
  <pollutionIncident> $no,
     (FOR $e in $recs
      WHERE $e/date > "1/1/90" and contains($e/diagnosis, "pollution")
      RETURN $e/diagnosis)
  </pollutionIncident>
```

The variable `$recs`, defined in the `LET` clause, is bound to the whole value of the expression assigned to it. FLWR expressions allow combining data from different documents, grouping, construction of new structure, and are natural candidates for query composition.

**Functions.** XQuery provides syntax for defining functions; these can be called in XQuery queries. The syntax for a function definition is illustrated in the following example: the recursive function `pastEntries` returns the list of all past record entries linked to a given record. This example also features a function from the standard library, `empty` (other functions like `count, max, avg` etc. are included).

```
FOR    $e in document("med.xml")//record[@ssNo="123"]/entry
WHERE  $e/date="1/1/1990"
RETURN pastEntries($e)

FUNCTION pastEntries(ELEMENT $e) RETURNS [ELEMENT] {
  IF empty ($e/@rel_previous->entry)
     THEN []
     ELSE ($e/@rel_previous->entry UNION pastEntries($e/@rel_previous->entry)
}
```

XQuery can dynamically integrate external data sources via the `document(URL)` function. The `document(URL)` function can be used to process the XML results of remote operations invoked over HTTP. For example, given a service description with a `getNetworkLoad()` operation, a query can match on values dynamically produced by that operation.

### Simple, Medium and Complex Queries

The suitability of the query language for service and resource discovery is now demonstrated by formulating example prose queries in the language.

**Simple Query.** Example simple queries are:

- *(QS1) Find all (available) services.*

```
RETURN /tupleset/tuple[@type="service"]
```

- *(QS2) Find all services that implement a replica catalog service interface and that CMS members are allowed to use, and that have an HTTP binding for the replica catalog operation "XML getPFNs(String LFN).*

```
LET $repcat := "http://gridforum.org/interface/ReplicaCatalog-1.0"
FOR $tuple in /tupleset/tuple[@type="service"]
LET $s := $tuple/content/service
WHERE
  SOME $op IN $s/interface[@type = $repcat]/operation SATISFIES
    ($op/name="XML getPFNs(String LFN)" AND $op/bindhttp/@verb="GET"
     AND contains($op/allow, "http://cms.cern.ch/everybody"))
RETURN $tuple
```

- *(QS4) Find all local services (all service interfaces of any given service must reside on the same host).*

```
FOR $tuple in /tupleset/tuple[@type="service"]
LET $s := $tuple/content/service
WHERE count(distinct(hostname($s/interface/operation/bindhttp/@URL))) <= 1
RETURN $tuple
```

- *(QS5) Find all services and return their service links (instead of descriptions).*

```
FOR $tuple in $doc/tupleset/tuple[@type="service"]
RETURN <tuple> {$tuple/attribute::*} </tuple>
```

- *(QS6) Find all CMS replica catalogs and return their physical file names (PFNs) for a given logical file name (LFN); suppress PFNs not starting with "ftp://".*

```
LET $repcat := "http://gridforum.org/interface/ReplicaCatalog-1.0"
FOR $tuple in /tupleset/tuple[@type="service"]
LET $s := $tuple/content/service
WHERE
  SOME $op IN $s/interface[@type = $repcat]/operation SATISFIES
    ($op/name="XML getPFNs(String LFN)" AND $op/bindhttp/@verb ="GET"
     AND contains($op/allow, "http://cms.cern.ch/everybody"))
RETURN
  FOR $pfn IN invoke($s, $repcat, "XML getPFNs(String LFN)",
    "http://myhost.cern.ch/myFile")/tupleset/PFN
  WHERE starts-with($pfn, "ftp://")
  RETURN $pfn
```

**Medium Query.** Example medium queries are:

- *(QM1) Find the CMS storage service with the largest network bandwidth to my host "dummy.cern.ch" (assuming there exists a service estimating bandwidth from A to B).*

```
LET $source := "dummy.cern.ch"
LET $storage := "http://gridforum.org/interface/storage-1.0"
LET $sorted := /tupleset/tuple[@type="service" AND content/service/@owner="cms.org" AND
  content/service/interface/@type=$storage]
  SORTBY (bandwidth($source, host(./@link)))
RETURN $sorted[last()]

DEFINE FUNCTION bandwidth($source, $dest) {
  document("http://netestimator.cern.ch/estimate?source=", $source, "&dest=", $dest)
}
```

- *(QM2) Return the number of replica catalogs services.*

```
LET $repcat := "http://gridforum.org/interface/ReplicaCatalog-1.0"
RETURN count(/tupleset/content/service[interface/@type=$repcat])
```

- *(QM3) Find the two CMS execution services with minimum and maximum CPU load and return their service description and load.*

```
LET $executor := "http://gridforum.org/interface/executor-1.0"
LET $tuples := /tupleset/tuple[@type="service" AND content/service/@owner="cms.org"
                               AND content/service/interface/@type=$executor]]
LET $sorted := FOR $tuple IN $tuples RETURN
  <item>
    {$tuple}
    <load>
      {invoke($tuple/content/service, $executor, "String cpuLoad()", "")}
    </load>
  </item> SORTBY (load)
RETURN
  <min> $sorted[1] </min>
  <max> $sorted[last()] </max>
```

- *(QM4) Return a summary of all replica catalogs and schedulers residing within the domains "cern.ch", "infn.it" and "anl.gov", grouped in ascending order by owner, domain and service type, with aggregate group cardinalities. A sample result set should look as follows:*

```
<tupleset>
  <owner name="alice.org" domainSize="2">
    <domain name="cern.ch" typeSize="2">
      <type name="http://gridforum.org/interface/ReplicaCatalog-1.0"
          serviceSize="13" />
      <type name="http://gridforum.org/interface/Scheduler-1.0"
          serviceSize="4" />
    </domain>
    <domain name="infn.it" typeSize="1">
        ...
```

```
    </domain>
  </owner>
  <owner name="cms.org" domainSize="3">
      ...
  </owner>
</tupleset>
```

A suitable XQuery reads as follows:

```
LET $types := ("http://gridforum.org/interface/ReplicaCatalog-1.0",
  "http://gridforum.org/interface/Scheduler-1.0")
LET $alldomains := ("cern.ch", "infn.it", "anl.gov")
LET $services := /tupleset/tuple[@type="service"]/content/service[contains($alldomains,@domain)
  AND contains($types,interface/@type)]
FOR $owner IN distinct($services/@owner) SORTBY (@owner)
LET $domains := distinct($services[@owner=$owner]/@domain) SORTBY (@domain)
RETURN
  <owner name={$owner} domainSize={count($domains)}> {
    FOR $domain IN $domains
    LET $types := distinct($services[@owner=$owner AND
      @domain=$domain]/interface/@type) SORTBY (@type)
    RETURN
      <domain name={string($domain)} typeSize={count($types)}> {
        FOR $type in $types
        LET $srvs := $services[@owner=$owner AND @domain=$domain AND interface/@type=$type]
        RETURN
          <type name={string($type)} serviceSize={count($srvs)}/>
        }
      </domain>
  }
  </owner>
```

**Complex Query.** Example complex queries are:

- *(QC1) Find all (execution service, storage service) pairs where both services of a pair live within the same domain. (Job wants to read and write locally).*

```
LET $exeType := "http://gridforum.org/interface/executor-1.0"
LET $stoType := "http://gridforum.org/interface/storage-1.0"
FOR $executor IN /tupleset/tuple[content/service/interface/@type = $exeType],
    $storage  IN /tupleset/tuple[content/service/interface/@type = $stoType AND
    domainName(@link) = domainName($executor/@link)]
RETURN
  <pair>
    {$executor}
    {$storage}
  </pair>
```

- *(QC2) Find all hosts that run more than one replica catalog with CMS as owner. (Want to check for anomalies).*

```
LET $repcat := "http://gridforum.org/interface/ReplicaCatalog-1.0"
LET $hosts := /tupleset/tuple/hostname(@link)[content/service[interface/@type = $repcat AND
                                                content/service/@owner = "cms.org"]]
FOR host IN $hosts
RETURN <host> {$host} </host>
WHERE count(hosts[./ = $host]) > 1
```

## 3.6 Related Work

Searching deep, inter-related and/or complex data structures typically requires a powerful query language. We estimate that WSDL based service descriptions are about one order of magnitude larger in size and structural complexity than corresponding SWSDL based descriptions (see Section 2.3). Consequently, the example queries posed in this chapter are often trivial in comparison with realistic queries on WSDL data structures. Let us assess the suitability of the query capabilities of various query languages in the context of service and resource discovery.

**LDAP.** The Lightweight Directory Access Protocol (LDAP) [14] inherits its query and data model from X.500 [13]. The data model is not dynamic and it is not XML based. No example service discovery query except *QS1* and *QS5* can be expressed with the LDAP query language. This would also be the case if LDAP were defined on an XML data model. An example system using this query and data model is the Metacomputing Directory Service (MDS) [15, 16].

The data model is based on the *entry*, which contains data about some object (e.g. a person). An entry is composed of attributes, which have a type and one or more values. The attribute type determines what kinds of values are legal. An entry has a mandatory attribute, which is a hierarchical identifier termed *distinguished name* (*DN*). An example DN is `cn=Barbara Jensen, o=University of Michigan, c=US`. Because of its hierarchical nature, a DN can be seen as organizing a set of entries into a tree structure, the *Directory Information Tree* (DIT). Usually the tree is organized according to political, geographical, or organizational boundaries. For comparison, an HTTP URL with the usual attribute-value pairs can be considered equivalent to a DN. An XML tuple with a content link corresponds to an LDAP entry. A set of such tuples corresponds to a set of LDAP entries. Both sets can be interpreted as a tree. Operations are provided to query, add, modify, and delete entries from the tree.

The LDAP query language has the following capabilities. A query returns a set of matching entries. A query can specify a base DN, scope, filter, timeout, maximum result set size and the names of attributes to return for each matching entry. The base DN decides the position in the name space tree at which the search should be started. An empty string implies starting at the root of the tree. The scope flag indicates which entries should be considered: just the base DN entry, all immediate descendent entries of the DN, or all entries at or below the DN. The filter is applied to each entry selected by the scope. A filter is an expression that logically compares (`=, <=, >=`) the string value of an attribute (`email`) with a string constant, optionally with a substring match joker (`picture*.jpg`) and approximate

string equality test (∼). Filters can be combined with Boolean `AND, OR` and `NOT` operators. For example, the query `o=anl.gov,c=US??persons?(&(cn=Mark*)(sn=G*))` returns every person entry whose name starts with Mark and whose surname starts with "G".

Clearly the expressive power of this language is insufficient for service and resource discovery use cases and most other non-trivial questions.

**SQL.** The relational data model is well suited for static centralized systems. However, it is unsuitable for a large distributed system spanning many administrative domains, populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources. The relational data model does not allow for semi-structured data. All tuples must be homogenously structured in the sense that their column values comply with a strongly typed schema. Tuples with different schema belong to different tables. Hence, a query cannot operate on a single (logical) set containing *all* tuples. A query must have out-of-band knowledge of the relevant table names and schemas, which themselves may not be heterogeneous but must be static, globally standardized and synchronized. This seriously limits the applicability of the relational model in the context of autonomy, decentralization, unreliability and frequent change.

SQL [19] is a rich and expressive general-purpose language defined over the relational data model. In addition to the above limitations, SQL lacks hierarchical navigation as a key feature and other capabilities such as dynamic data integration, expression nesting with full generality as well as regular expression matching. As a result, some example queries cannot be expressed (e.g. *QS6, QM1, QM3*) and most can only be expressed with extremely complex queries over a large number of auxiliary tables, as exemplified by Figure 3.2. The same holds for inserts and updates.

The relational data model and SQL are, for example, used in the Relational Grid Monitoring Architecture (RGMA) system [61] and the Unified Relational GIS Project [62].

**Other.** None of the example discovery queries can be satisfied with a lookup by key (e.g. globally unique name). This is the type of query assumed in most P2P systems such as DNS [20], Gnutella [21], Freenet [22], Tapestry [23], Chord [24] and Globe [25], leading to highly specialized *content-addressable* networks centered around the theme of distributed hash table lookup. Note further that almost no queries are exact match queries (i.e. given a flat set of attribute values find all tuples that carry exactly the same attribute values), assumed in systems such as SDS [26] and Jini [63]. They are also not fuzzy keyword searches, as used in web search engines. Next, queries do not specify that at most one result should be returned.

The limited expressiveness of the above mentioned query languages allows for easy implementation and some straightforward optimizations, but it also dramatically limits their applicability and ability to cope with changing requirements, leading to a flurry of very similar but not identical special-purpose systems, each supporting yet another narrow custom query type. These systems may well serve a special-purpose important for a given niche, but are unsuitable for supporting service discovery, let alone ubiquitous service and resource discovery for a wide range of applications and user communities.

The greater the number and heterogeneity of content and applications, the more important expressive general-purpose query capabilities become. Clearly realistic ubiquitous service and resource discovery *stands and falls* with the ability to express queries in a rich general-purpose query language. More precisely, a query language suitable for service and resource discovery should meet the requirements stated in Table 3.1 (in decreasing order of significance). As can be seen from the table, LDAP, SQL and XPath do not meet a number of essential requirements, whereas the XQuery language meets all requirements and desiderata posed.

| **Capability** | **XQuery** | **XPath** | **SQL** | **LDAP** |
|---|---|---|---|---|
| Simple, medium and complex queries over a set of tuples | yes | no | yes | no |
| Query over structured and semi-structured data | yes | yes | no | yes |
| Query over heterogeneous data | yes | yes | no | yes |
| Query over XML data model | yes | yes | no | no |
| Navigation through hierarchical data structures (Path Expressions) | yes | yes | no | exact match only |
| Joins (combine multiple data sources into a single result) | yes | no | yes | no |
| Dynamic data integration from multiple heterog. sources such as databases, documents and remote services | yes | yes | no | no |
| Data restructuring patterns (e.g. SELECT-FROM-WHERE in SQL) | yes | no | yes | no |
| Iteration over sets (e.g. FOR clause) | yes | no | yes | no |
| General-purpose predicate expressions (WHERE clause) | yes | no | yes | no |
| Nesting several kinds of expressions with full generality | yes | no | no | no |
| Binding of variables and creating new structures from bound variables (LET clause) | yes | no | yes | no |
| Constructive queries | yes | no | no | no |
| Conditional expressions (IF ... THEN ... ELSE) | yes | no | yes | no |
| Arithmetic, comparison, logical and set expressions | yes, all | yes | yes, all | log. & string |
| Operations on data types from a type system | yes | no | yes | no |
| Quantified expressions (e.g. SOME, EVERY clause) | yes | no | yes | no |
| Standard functions for sorting, string, math, aggregation | yes | no | yes | no |
| User defined functions | yes | no | yes | no |
| Regular expression matching | yes | yes | no | no |
| Concise and easy to understand queries | yes | yes | yes | yes |

Table 3.1: Capabilities of XQuery, XPath, SQL and LDAP query languages.

## 3.7 Summary

This chapter develops a database and query model as well as a generic and dynamic data model that address the problems of large heterogeneous distributed system spanning many administrative domains. A framework is used where there exist one or more autonomous nodes, each holding a database. The databases of nodes may be deployed in any arbitrary way according to some deployment model. For example, the databases of all nodes may be stored next to each other on a single central data server. The set of tuples in the universe is partitioned over the nodes. A *link topology* describes the link structure among nodes, which

**XQuery**

```
FOR x in document("med.xml")/medical/patient,
    y in document("med.xml")//patientSSno,
    z in x/name
WHERE x/@ssno=y
RETURN z
```

**SQL**

```
SELECT e3.elID as $z
FROM Document d1, URI u1, Value v1, Element e1, QName q1, Value v2, Child c1, Element e2,
     QName q2, Value v3, Attribute a1, Value v4, Value v5, Child c2, Element e3, QName q3,
     Value v6, TransClosure tc1, Element e4, QName q4, Value v7, Child c3, Value v8
WHERE d1.docURIID=u1.uriID and u1.uriValID=v1.valID and v1.value="med.xml" and
      d1.rootElemId=e1.elID and e1.elQNameID=q1.qNameID and q1.qnLocalID=v2.valID and
      v2.value="medical" and c1.parentID=e1.elID and c1.childID=e2.elID and
      e2.elQNameID=q2.qNameID and q2.qnLocalID=v3.valID and v3.value="patient" and
      a1.attrElID=e2.elID and a2.attrNameID=v4.valID and v4.value="SSno" and
      a1.attrValID=v5.valID and c2.parentID=e2.elID and c2.childID=e3.elID and
      e3.elQNameID=q3.qNameID and q3.qnLocalID=v6.valID and v6.value="name" and
      d1.rootElemID=tc2.parentID and tc2.childID=e4.elID and e4.elQameID=q4.qNameID and
      q4.qnLocalID=e7.valueID and v7.value="patientSSno" and c3.parentID=e3.elID and
      c3.childValID=v8.valID and v5.value=v8.value
```

Figure 3.2: Translation of Hierarchy Navigating XQuery to SQL [48].

may be arbitrary. For example, in a service discovery system, a link topology such as ring, tree or graph could tie together a distributed set of administrative domains, each hosting a registry node holding descriptions of services local to the domain.

Unlike in the relational model the elements of a tuple in our data model can hold structured or semi-structured data in the form of any arbitrary well-formed XML document or fragment. An individual tuple element may, but need not, have a schema, in which case the element must be valid according to the schema. The elements of all tuples may, but need not, share a common schema. A tuple is a multi-purpose data container that may contain arbitrary content.

The general-purpose query model is for read-only search and operates on *tuples*. The concepts of (logical) query and (physical) query scope are cleanly separated rather than interwoven. A query is formulated against a global database view and is insensitive to link topology and deployment model. In other words, to a query the set of all tuples appears as a single homogenous database, even though the set may be (recursively) partitioned across many nodes and databases. This means that in a relational or XML environment, at the global level, the set of all tuples appears as a single, very large, table or XML document, respectively. Unlike in a RDBMS, a single (logical) tuple set contains *all* tuples. This implies that a query need not specify a "table" or "tuple set name" to indicate the type of tuples that should be considered. Rather, predicates within the regular query language are used to select the desired tuples from the single set. The query scope, on the other hand, is used to navigate and prune the link topology and filter on attributes of the deployment model. A query is evaluated against a set of tuples. The set, in turn, is specified by the scope. Conceptually, the scope is the input fed to the query. The query scope is a set and may contain anything from all tuples in the universe to none. A scope can be specified directly or indirectly. The data model is XML based and models a document as a tree of nodes.

Example service discovery queries are given in prose. Three query types are identified, namely *simple, medium* and *complex*. A simple query finds all tuples matching a given predicate or pattern. A medium query computes an answer over a set of tuples as a whole, allowing for use of aggregation functions. A complex query also computes an answer over a set of tuples as a whole. However, it has powerful capabilities to combine data from multiple sources and elements, for example through joins. Complex queries are inefficient to answer without centralized database architectures.

An appropriate query language (XQuery) is suggested. The suitability of the query language is demonstrated by formulating the example prose queries in the language. Detailed requirements for a query language supporting service and resource discovery are given. The capabilities of various query languages are compared.

# Chapter 4

# A Database for Discovery of Distributed Content

## 4.1 Introduction

In a distributed system, it is desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. This enables information discovery and collective collaborative functionality that operate on the system as a whole, rather than on a given part of it. For example, it allows a search for descriptions of services of a file sharing system, to determine its total download capacity, the names of all participating organizations, etc. Example systems include a service discovery system, an electronic market place, or an instant messaging and news service. In all these systems, a variety of information describes the state of autonomous remote participants residing within different administrative domains. Participants frequently join, leave and act on a best effort basis. Discovery attempts to determine the global state of a distributed system. In a large distributed system spanning many administrative domains, predictable, timely, consistent and reliable global state maintenance is infeasible. The information to be aggregated and integrated may be outdated, inconsistent, or not available at all. Failure, misbehavior, security restrictions and continuous change are the norm rather than the exception. The key problem then is:

- *How should a database node maintain information populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources? In particular, how should it do so without sacrificing reliability, predictability and simplicity? How can powerful queries be expressed over time-sensitive dynamic information?*

In this chapter, a design and specification for a centralized type of database is developed that is conditioned to address the problem, the so-called *hyper registry*. The hyper registry is a general-purpose XQuery database with mechanisms for content caching and soft state maintenance. The hyper registry can maintain hyperlinks and cache content pointed to by these links. A content provider can publish a hyperlink, which in turn enables the hyper registry (and third parties) to pull (retrieve) the current content. Optionally, a content provider can also include a copy of the current content as part of publication. The hyper registry may support caching, for example in server pull or client push mode or both. For reliable, predictable and simple distributed state maintenance, hyper registry tuples are maintained

as *soft state*. To condition for overload, limit resource consumption, and satisfy minimum requirements on content freshness, mechanisms to *throttle* refresh and query frequency are proposed, adaptively inviting more or less traffic over time. Content link, content cache, a hybrid pull/push communication model and the expressive power of XQuery allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client.

**Overall Model.** A *hyper registry* has a database that holds a set of tuples. A *tuple* is an annotated multi-purpose soft state data container that may contain arbitrary piece of *content*. Examples for content include a service description formulated in SWSDL or WSDL [8], a file, picture, current network load, host information, stock quotes, etc. Example content is a service description that reads:

```
<service>
  <interface type = "http://gridforum.org/interface/Scheduler-1.0">
    <operation>
      <name>void submitJob(String jobdescription)</name>
      <allow> http://cms.cern.ch/everybody </allow>
      <bind:http verb="GET" URL="https://sched.cern.ch/scheduler/submitjob"/>
    </operation>
  </interface>
</service>
```

A second example content is a list of host system information such as:

```
<hostInfo>
   <host name="fred01.cern.ch" os="redhat 7.2" arch="i386" mem="512M" MHz="1000" cpus="2"/>
   <host name="fred02.cern.ch" os="solaris 2.7" arch="sparc" mem="8192M" MHz="400" cpus="64"/>
</hostInfo>
```

A remote client can query a hyper registry in a query language, obtaining a set of tuples as answer. In a given context, a content provider can publish content of a given type to one or more hyper registries. More precisely, a *content provider* can publish a dynamic pointer called a *content link*, which in turn enables the hyper registry and third parties to retrieve (pull) the current content presented from the provider at any time. Optionally, a content provider can also include a copy of the current content as part of publication (push). In any case, a hyper registry may support caching of content.

Content *caching* is important for client efficiency. The hyper registry may not only keep links but also a copy of the current content pointed to by the link. With caching, clients no longer need to establish a network connection for each content link in a query result set in order to obtain content. This avoids prohibitive latency, in particular in the presence of large result sets. A hyper registry may (but need not) support caching, for example in server pull or client push mode or both.

For reliable, predictable and simple distributed state maintenance, a hyper registry tuple is maintained as *soft state*. A tuple may eventually be discarded unless refreshed by a stream of timely confirmation notifications from the content provider [37]. To this end, a tuple carries timestamps. A tuple is expired and removed unless explicitly renewed via timely periodic

publication, henceforth termed *refresh*. In other words, a refresh allows a content provider to cause a content link and/or cached content to remain present for a further time.

To condition for overload, limit resource consumption and satisfy minimum requirements on content freshness, mechanisms to *throttle* refresh and query frequency are proposed, adaptively inviting more or less traffic over time.

The entry barrier to participation in the system should be very low. To this end, the system is designed to be as simple as possible at the edges of the network. A content provider is cleanly decoupled from the hyper registry and only requires the ubiquitous HTTP protocol for communication with a local or remote hyper registry. For example, it may be sufficient to run an Apache server [64] and `cron` job to publish content[1].

## 4.2 Content Link and Content Provider

**Content Link.** A *content link* is an HTTP(S) URL pointing to the content of a content provider. An HTTP(S) GET request to the link must return the current content, subject to local security policy[2]. In other words, a simple hyperlink is employed. In the context of service discovery, we use the term *service link* to denote a content link that points to a service description. Like in the WWW, content links can freely be chosen as long as they conform to the HTTP URL specification [65]. Hence, they may contain the usual URL encoded attribute-value pairs. The semantics of the structure of a given content link are opaque to a client. Examples for content links are:

```
http://sched.cern.ch:8080/getServiceDescription.wsdl
https://cms.cern.ch/getServiceDescription?id=4712&cache=disable
http://sched.infn.it/getHostInfo
http://phone.cern.ch/lookup?query="select phone from book where phone=4711"
http://repcat.cern.ch/getPhysicalFileNames?lfn="myLogicalFileName"
```

**Content Provider.** A *content provider* offers information conforming to a homogeneous global data model. In order to do so, it typically uses some kind of internal mediator to transform information from a local or proprietary data model to the global data model. A content provider can be seen as a gateway to heterogeneous content sources. The global data model is the Dynamic Data Model (DDM) introduced in Section 3.3. Hence, content can be structured or semi-structured data in the form of any arbitrary well-formed XML [11] document or fragment. Individual content may, but need not, have a schema (XML Schema [12]), in which case content must be valid according to the schema. All content may, but need not, share a common schema. This flexibility is important for integration of heterogeneous content.

A content provider is an umbrella term for two components, namely a presenter and a publisher. The *presenter* is a service and answers HTTP(S) GET content retrieval requests from a hyper registry or client (subject to local security policy). The *publisher* is a piece of

[1] `cron` is a standard Unix daemon that periodically executes programs or shell scripts according to flexible configuration data.

[2] HTTP 1.1 should be supported to allow for efficient persistent connections.

code that publishes content link, and perhaps also content, to a hyper registry. The publisher need not be a service, although it uses HTTP(S) POST for transport of communications. The structure of a content provider and its interaction with a hyper registry and a client are depicted in Figure 4.1 (a). Note that a client can bypass a hyper registry and directly pull current content from a provider.

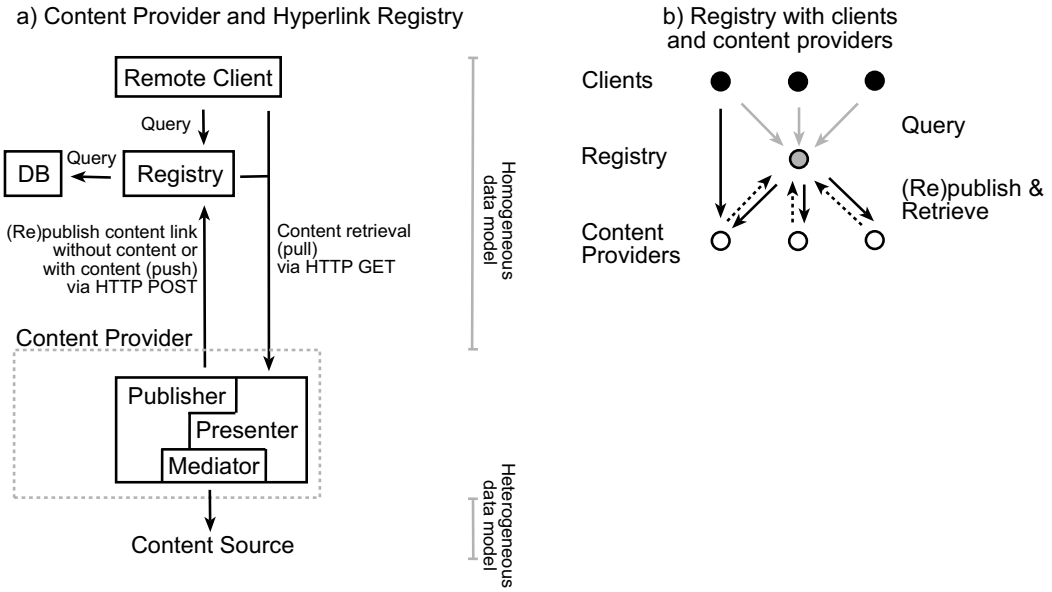


Figure 4.1: Content Provider and Hyper Registry.

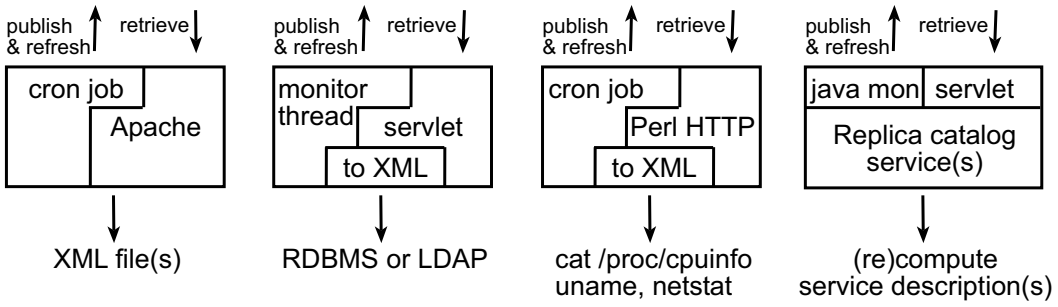


Figure 4.2: Example Content Providers.

Just as in the dynamic WWW that allows for a broad variety of implementations for the given protocol, it is left unspecified how a presenter computes content on retrieval. Content can be static or dynamic (generated on the fly). For example, a presenter may serve the content directly from a file or database, or from a potentially outdated cache. For increased accuracy, it may also dynamically recompute the content on each request.

Consider the examples in Figure 4.2. A simple but nonetheless very useful content provider uses a commodity HTTP server such as Apache to present XML content from the file system.

A simple `cron` job monitors the health of the Apache server and publishes the current state to a hyper registry. Another example for a content provider is a Java servlet that makes available data kept in a relational or LDAP database system. A content provider can execute legacy command line tools to publish system state information such as network statistics, operating system and type of CPU. Another example for a content provider is a network service such as a replica catalog that (in addition to servicing replica lookup requests) publishes its service description and/or link so that clients may discover and subsequently invoke it.

Figure 4.1 (b) illustrates a hyper registry with several content providers and clients. Providers and hyper registries can be deployed and configured arbitrarily. For example, in a strategy for scalable administration of large cluster environments, a single shared Apache web server can easily be configured to serve XML descriptions of thousands of services on hundreds of hosts. For example, via a naming convention we can assign a distinct web server directory and corresponding service link for each host-service combination. To serve descriptions it is sufficient to have some administrative `cron` job run periodically on each service host, which writes the current service description into an appropriate XML file in the appropriate directory on the web server.

Presenter and publisher are not required to run in the same process or even on the same host. For efficiency, a so-called *container* of a virtual hosting environment may be used to run more than one provider in the same process or thread. For example, a highly efficient and scalable container such as the Apache Tomcat servlet engine [66] not only can serve many hundreds to a thousand (light-weight) concurrent requests per second on a commodity PC, but it can also embed any number of dynamic provider types in the same process, with each provider type being capable of serving any number of provider instances. Typically, a provider is remote to the hyper registry. This is not a requirement, however. A provider may also be local to the hyper registry and connect through a loop-back connection. For further efficiency, a hyper registry may internally host any number of built in providers. Using commodity servlet technology, any number of hyper registries can be hosted within the same container.

## 4.3 Publication

In a given context, a content provider can publish content of a given type to one or more hyper registries. More precisely, a content provider can publish a dynamic pointer called a content link, which in turn enables the hyper registry (and third parties) to retrieve the current content. For efficiency, the `publish` operation takes as input a set of zero or more tuples. Each input tuple has a content link, a type, a context, some time stamps, and (optionally) arbitrary metadata and content extensibility elements. Consider the following example tuple.

| **Link** | **Context** | **Type** | **TS1** | **TC** | **TS2** | **TS3** | **Metadata** | **Content** |
|---|---|---|---|---|---|---|---|---|
| http://sched001.cern.ch/ getServiceDescription | Parent | Service | 10 | 10 | 20 | 30 | <owner name = "http://cern.ch"/> | <service> ... < /service> |

A tuple is an annotated multi-purpose soft state data container that may contain a piece of arbitrary content and allows for refresh of that content at any time, as depicted in Figure

4.3.

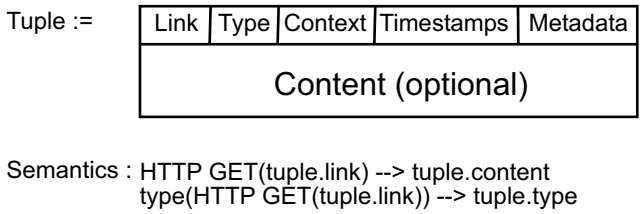


Figure 4.3: Tuple Link allows for Refresh of Tuple Content at any time.

- **Link.** The content link is an HTTP(S) URL as introduced above. Given the link the current content of a content provider can be retrieved (pulled) at any time.

- **Type.** The type describes *what* kind of content is being published (e.g. `service`, `application/octet-stream`, `image/jpeg`, `networkLoad`, `hostinfo`).

- **Context.** The context describes *why* the content is being published or *how* it should be used (e.g. `child, parent, x-ireferral, gnutella, monitoring`). For example, the `parent` and `child` context will later be used for top down query routing through node topologies with explicit hierarchical structure. Note that it is often not meaningful to embed context information in the content itself, because a given content can be published in a different context at different hyper registries (1:N association). For example, a service can be a child of some nodes and at the same time be a parent for some other nodes. However, its service description (content) should clearly remain invariant. In addition, context and type allow a query to differente on crucial attributes even if content caching is not supported or not authorized[3].

- **Timestamps TS1, TS2, TS3, TC.** Discussion of timestamps is deferred to Section 4.6 below.

- **Metadata.** The metadata element further describes the content and/or its retrieval beyond what can be expressed with the previous attributes. For example, the metadata may be a secure digital signature [68] of the content. It may describe the authoritative content provider or owner of the content. Another metadata example is a Web Service Inspection Language (WSIL) document [69] or fragment thereof, specifying additional

[3]For clarity of exposition, the examples given here use short strings as values for type and context (e.g. `service, parent`). However, strictly speaking, this is not legal. To avoid namespace pollution and ambiguities, the value of a type must be a universally unique URI [65], a MIME content-type [67] (e.g. `application/octet-stream, image/jpeg, audio/mpeg`) or the empty string. For XML content observing an XML Schema [12] the type must equal the URI of the schema namespace. The value of a context must be a URI or the empty string. For example, a service description really is of type `http://gridforum.org/content-type/service-1.0` whereas the `parent` context really has the value `http://gridforum.org/content-context/parent-1.0`. The idea is not new. For example, HTTP, SMTP, XML namespaces and schema types [12] employ the same approach towards identifiers.

content retrieval mechanisms beyond HTTP content link retrieval. The metadata argument may be any well-formed XML document or fragment. It is an extensibility element enabling customization and flexible evolution. It is optional.

- **Content.** Given the link the current content of a content provider can be retrieved (pulled) at any time. Optionally, a content provider can also include a copy of the current content as part of publication (push). For clarity of exposition, the published content is an XML element[4].

According to the Dynamic Data Model defined in Section 3.3, an XML representation for the set of tuples is used. Consider the following example tuple set:

```
<tupleset>
    <tuple link="http://registry.cern.ch/getServiceDescription"
           type="service" ctx="parent" TS1="10" TC="15" TS2="20" TS3="30">
       <content>
         <service>
           <interface type ="http://gridforum.org/interface/Presenter-1.0">
             <operation>
               <name>XML getServiceDescription()</name>
               <bind:http verb="GET" URL="https://registry.cern.ch/getServiceDescription"/>
             </operation>
           </interface>

           <interface type = "http://gridforum.org/interface/XQuery-1.0">
             <operation>
               <name> XML query(XQuery query)</name>
               <bind:beep URL="beep://registry.cern.ch:9000"/>
             </operation>
           </interface>
         </service>
       </content>
       <metadata>
         <owner name="http://cms.cern.ch"/>
       </metadata>
    </tuple>

    <tuple link="http://repcat.cern.ch/getServiceDescription?id=4711"
           type="service" ctx="child" TS1="30" TC="0" TS2="40" TS3="50">
    </tuple>
</tupleset>
```

The publish operation of a hyper registry has the following signature:

```
void publish(XML tupleset)
```

[4]In the general case (allowing non-text based content types such as `image/jpeg`), the content is a MIME object. The XML based publication input set and query result set is augmented with an additional MIME multipart object [67], which is a list containing all content. The content element of the result set is interpreted as an index into the MIME multipart object. A typical hyper registry that supports caching can handle content with at least MIME content-type `text/xml` and `text/plain`.

Within a tuple set, a tuple is uniquely identified by its *tuple key*, which is the pair `(content link, context)`. If a key does not already exist on publication, a tuple is inserted into the hyper registry database. An existing tuple can be updated by publishing other values under the same tuple key. An existing tuple (key) is "owned" by the content provider that created it with the first publication. It is recommended that a content provider with another identity must not be permitted to publish or update the tuple.

## 4.4 Query

**Minimalist Query.** As a minimum, clients can query the hyper registry by invoking minimalist query operations such as `getTuples`. The `getTuples` query operation takes no arguments and returns the full set of all published tuples "as is". That is, query output format and publication input format are the same. If supported, output includes cached content. An example result set for a query reads (discussion of timestamps attributes `TS` is again deferred to Section 4.6 below):

```
<tupleset TS4="100">
    <tuple link="http://sched001.cern.ch/getServiceDescription"
           type="service" ctx="parent" TS1="10" TC="15" TS2="20" TS3="30">
       <content>
           <service> service description A goes here </service>
       </content>
       <metadata>
           <owner name="http://cms.cern.ch"/>
       </metadata>
    </tuple>

    <tuple link="http://repcat.cern.ch/pub/getServiceDescription?id=4711"
           type="service" ctx="child" TS1="30" TC="0" TS2="40" TS3="50">
    </tuple>

    <tuple link="http://repcat.cern.ch/pub/getStatistics"
           type="repcatStats" TS1="60" TC="65" TS2="70" TS3="80">
       <content>
           <repcatStats host="repcat.cern.ch" avgHitsPerMin="1000">
              <dbsize countLFNs="100000" countPFNs="100000000"/>
           </repcatStats>
       </content>
    </tuple>

    <tuple link="http://monitor.cern.ch/pub/getHostInfo"
           type="hostInfo" TC="65" TS1="60" TS2="70" TS3="80">
       <content>
           <hostInfo>
               <host name="fred01.cern.ch" os="redhat 7.2" arch="i386"
                     mem="512M" MHz="1000" cpus="2"/>
               <host name="fred02.cern.ch" os="solaris 2.7" arch="sparc"
                     mem="8192M" MHz="400" cpus="64"/>
           </hostInfo>
       </content>
```

```
    </tuple>
</tupleset>
```

The `getLinks` query operation is similar in that it also takes no arguments and returns the full set of all tuples. However, it always substitutes an empty string for cached content. In other words, the content is omitted from tuples, potentially saving substantial bandwidth. The second tuple in the example above has such a form.

**XQuery.** Clearly many kinds of sophisticated query capabilities can be introduced. Advanced query support can be expressed on top of the basic capabilities introduced so far. Here we assume that a node has the advanced capability to execute XQueries over the set of tuples it holds in its database. The XQuery language allows for powerful searching, which is critical for non-trivial applications. Example XQueries have already been given in Section 3.5. The same rules that apply to minimalist queries also apply to XQuery support. An implementation might use a modular and simple XQuery processor such as `Quip` [70] for an operation such as `XML query(XQuery query)`. Because not only content, but also content link, context, type, time stamps, metadata etc. are part of a tuple, a query can also select on this information.

**Deployment.** For flexibility, a hyper registry may be deployed in any arbitrary way (*deployment model*). For example, the database can be kept in a XML file, in the same format as returned by the `getTuples` query operation. However, tuples can certainly also be kept in a remote relational database (table), for example as follows:

| Link | Context | Type | TS1 | TC | TS2 | TS3 | Metadata | Content |
|---|---|---|---|---|---|---|---|---|
| http://sched001.cern.ch/getServiceDescription | Parent | Service | 10 | 15 | 20 | 30 | <owner name = "http://cms.cern.ch"/> | <service> A < /service> |
| http://sched.infn.it:8080/pub/getServiceDescription | Child | Service | 20 | 25 | 30 | 40 | null | <service> B < /service> |
| http://repcat.cern.ch/pub/getServiceDescription?id=4711 | Child | Service | 30 | 0 | 40 | 50 | null | null |
| http://repcat.cern.ch/pub/getStatistics | Null | RepStats | 60 | 65 | 70 | 80 | null | <repcatStats> ... </repcatStats> |

## 4.5 Caching

Content *caching* is important for client efficiency. The hyper registry may not only keep content links but also a copy of the current content pointed to by the link. With caching, clients no longer need to establish a network connection for each content link in a query result set in order to obtain content. This avoids prohibitive latency, in particular in the presence of large result sets. A hyper registry may (but need not) support caching. A hyper registry that does not support caching ignores any content handed from a content provider. It keeps content links only. Instead of cached content it returns empty strings. Cache coherency issues

arise. The query operations of a caching hyper registry may return tuples with stale content, i.e. content that is out of date with respect to its master copy at the content provider.

A caching hyper registry may implement a *strong* or *weak cache coherency policy*. A strong cache coherency policy is *server invalidation* [71]. Here a content provider notifies the hyper registry with a publication tuple whenever it has locally modified the content. We use this approach in an adapted version where a caching hyper registry can operate according to the client push pattern (*push hyper registry*) or server pull pattern (*pull hyper registry*) or a hybrid thereof. The respective interactions are as follows:

- **Pull Hyper Registry.** A content provider publishes a content link. The hyper registry then pulls the current content via content link retrieval into the cache. Whenever the content provider modifies the content, it notifies the hyper registry with a publication tuple carrying the time the content was last modified. The hyper registry may then decide to pull the current content again, in order to update the cache. It is up to the hyper registry to decide if and when to pull content. A hyper registry may pull content at any time. For example, it may dynamically pull fresh content for tuples affected by a query. This is important for frequently changing dynamic data such as network load.

- **Push Hyper Registry.** A publication tuple pushed from a content provider to the hyper registry contains not only a content link but also its current content. Whenever a content provider modifies content, it pushes the new content to the hyper registry, which may update the cache accordingly[5].

- **Hybrid Hyper Registry.** A hybrid hyper registry implements both pull and push interactions. If a content provider merely notifies that its content has changed, the hyper registry may choose to pull the current content into the cache. If a content provider pushes content, the cache may be updated with the pushed content. This is the type of hyper registry subsequently assumed whenever a caching hyper registry is discussed.

A non-caching hyper registry ignores content elements, if present. A publication is said to be *without content* if the content is not provided at all. Otherwise, it is said to be *with content*. Publication without content implies that no statement at all about cached content is being made (neutral). It does *not* imply that content should not be cached or invalidated.

A client must not assume that content is cached. For example, a hyper registry may not implement caching or it may be denied authorization when attempting to retrieve a given content, or it may ignore content provided on provider push. While it may be harmless

[5]If a push hyper registry does not also implement the pull pattern for hybrid behavior, the content link may (but need not) merely serve as an identifier for the content. As far as the hyper registry is concerned, the link need not be alive and point to meaningful content, because it never pulls anyway. The example host information content given previously may be identified by a unique but non-existing content link such as `http://fred.cern.ch/blabla`. The hyper registry may be used as a "dumb" repository into which content providers store information that cannot or should not be retrieved directly from the provider itself. In other words, the cache may be (ab)used as content store. It is meaningless to have a push hyper registry that does not implement "caching".

that potentially anybody can learn that some content exists (content link), stringent trust delegation policies may dictate that only a few select clients, not including the hyper registry, are allowed to retrieve the content from a provider. Consider for example, that a detailed service description may be helpful for launching well-focused security attacks. In addition, a hyper registry may have an authorization policy. For example, depending on the identity of a client, a registry may return a subset of the full result set or hide cached content and instead return the tuple substituted by empty content. Similarly, depending on content provider identity, pushed content may be ignored or rejected, or publication denied altogether.

## 4.6 Soft State

For reliable, predictable and simple distributed state maintenance, a hyper registry tuple is maintained as *soft state*. A tuple may eventually be discarded unless refreshed by a stream of timely confirmation notifications from a content provider. To this end, a tuple carries timestamps. A tuple is expired and removed unless explicitly renewed via timely periodic publication, henceforth termed *refresh*. In other words, a refresh allows a content provider to cause a content link and/or cached content to remain present a for a further time.

The strong cache coherency policy *server invalidation* is extended. For flexibility and expressiveness, the ideas of the Grid Notification Framework [37] are adapted. The publication operation takes four absolute time stamps `TS1, TS2, TS3, TC` per tuple[6].

The semantics are as follows. The content provider asserts that its content was last modified at time `TS1` and that its current content is expected to be valid from time `TS1` until at least time `TS2`. It is expected that the content link is alive between time `TS1` and at least time `TS3`. Time stamps must obey the constraint `TS1` $\leq$ `TS2` $\leq$ `TS3`. `TS2` triggers expiration of cached content, whereas `TS3` triggers expiration of content links. Usually, `TS1` equals the time of last modification or first publication, TS2 equals `TS1` plus some minutes or hours, and `TS3` equals `TS2` plus some hours or days. For example, `TS1`, `TS2` and `TS3` can reflect publication time, 10 minutes, and 2 hours, respectively.

A tuple also carries a timestamp `TC` that indicates the time when the tuple's embedded content (not the provider's master copy of the content) was last modified, typically by the last intermediary in the path between client and content provider (e.g. the hyper registry). If a content provider publishes with content, then we usually have `TS1=TC`. `TC` must be zero-valued if the tuple contains no content. Hence, a hyper registry not supporting caching always has `TC` set to zero. For example, a highly dynamic network load provider may publish its link without content and `TS1=TS2` to suggest that it is inappropriate to cache its content. Constants are published with content and `TS2=TS3=infinity, TS1=TC=currentTime`. These soft state time stamp semantics are summarized in Table 4.1.

Insert, update and delete of tuples occur at the timestamp-driven state transitions summarized in Figure 4.4. Within a tuple set, a tuple is uniquely identified by its *tuple key*, which

[6]To allow for meaningful comparison, sources generating time stamps and sinks processing time stamps must be synchronized, for example using the NTP network time protocol [72]. Further, they must share a common representation of time, for example as proposed in [73]. We subsequently assume a straightforward standard representation: the difference, measured in milliseconds, between the given UTC time and midnight, January 1, 1970 UTC.

| Time Stamp | Semantics |
|---|---|
| TS1 | Time content provider last modified content |
| TC | Time embedded tuple content was last modified (e.g. by an intermediary) |
| TS2 | Expected time while current content at provider is at least valid |
| TS3 | Expected time while content link at provider is at least valid (alive) |

Table 4.1: Soft State Time Stamp Semantics.

is the pair `(content link, context)`. A tuple can be in one of three states: *unknown*, *not cached*, or *cached*. A tuple is unknown if it is not contained in the hyper registry (i.e. its key does not exist). Otherwise, it is known. When a tuple is assigned *not cached* state, its last internal modification time `TC` is (re)set to zero and the cache is deleted, if present. For a *not cached* tuple we have `TC < TS1`. When a tuple is assigned *cached* state, the content is updated and `TC` is set to the current time. For a *cached* tuple, we have `TC` $\geq$ `TS1`.

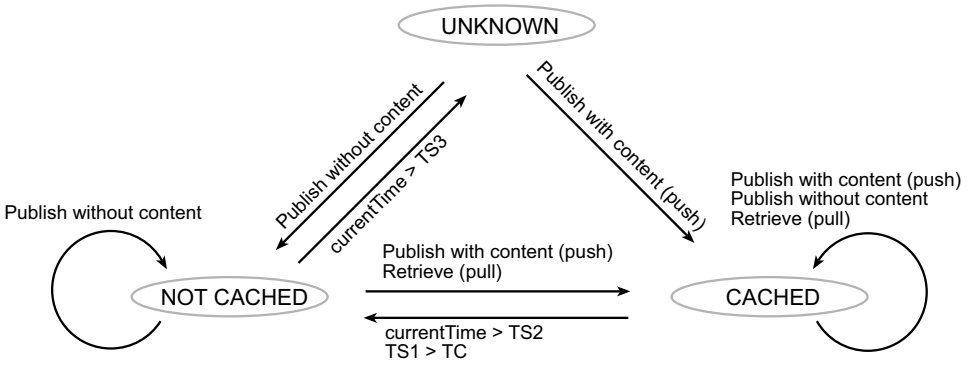


Figure 4.4: Soft State Transitions.

A tuple moves from *unknown* to *cached* or *not cached* state if the provider publishes with or without content, respectively. A tuple becomes *unknown* if its content link expires (`currentTime > TS3`); the tuple is then deleted. A provider can force tuple deletion by publishing with `currentTime > TS3`. A tuple is upgraded from *not cached* to *cached* state if a provider push publishes with content or if the hyper registry pulls the current content itself via retrieval. On content pull, a hyper registry may leave `TS2` unchanged, but it may also follow a policy that extends the lifetime of the tuple (or any other policy it sees fit). A tuple is degraded from *cached* to *not cached* state if the content expires. Such expiry occurs when no refresh is received in time (`currentTime > TS2`), or if a refresh indicates that the provider has modified the content (`TC < TS1`). The following pseudo-code illustrates publication and pull retrieval:

```
publish(..., TS1', TS2', TS3', content') :=
   if unknown then setContent(null)
   if content' != null then setContent(content')
   TS1 = TS1', TS2 = TS2', TS3 = TS3'
   trigger time driven state transitions ...
```

```
pull(content') :=
   setContent(content')
   if policy = "extend lifetime" then TS2 = currentTime() + 60 seconds
   trigger time driven state transitions ...

setContent(content') :=
   content = content'
   if content = null then state = not cached, TC = 0
                     else state = cached, TC = currentTime()
```

If a hyper registry receives multiple relevant publication tuples about a particular tuple key (i.e. `(content link, context)` pair), only the tuple with the most recent `TS1` is considered (update). The other tuples referring to the key are discarded. Finally, note that strictly speaking, a hyper registry may interpret time stamps as hints rather than orders. This allows avoiding abuse by providers, for example via infinite time stamps.

## 4.7 Flexible Freshness

Content link, content cache, a hybrid pull/push communication model and the expressive power of XQuery allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client. All three components may indicate how to manage content according to their respective notions of freshness.

For example, a content provider can model the freshness of its content via pushing appropriate timestamps and content. A hyper registry can model the freshness of its content via controlled acceptance of provider publications and by actively pulling fresh content from the provider. If a result (e.g. network statistics) is up to date according to the hyper registry, but out of date according to the client, the client can pull fresh content from providers as it sees fit. However, this is inefficient for large result sets. Nevertheless, it is important for clients that query results are returned according to their notion of freshness, in particular in the presence of frequently changing dynamic content.

Recall that it is up to the hyper registry to decide to what extent its cache is stale, and if and when to pull fresh content. For example, a hyper registry may implement a policy that dynamically pulls fresh content for a tuple whenever a query touches (affects) the tuple. For example, if a query interprets the content link URL as an identifier within a hierarchical name space (e.g. as in LDAP) and selects only tuples within a sub-tree of the name space, only these tuples should be considered for refresh. The more powerful a query language, the more complex is the required logic for query parsing, analysis, decomposition and merging, etc.

**Refresh-on-client-demand.** So far, a hyper registry must guess what a client's notion of freshness might be, while at the same time maintaining its decisive authority. A client still has no way to indicate (as opposed to force) its view of the matter to a hyper registry. We propose to address this problem with a simple and elegant *refresh-on-client-demand* strategy

under control of the hyper registry's authority. The strategy exploits the rich expressiveness and dynamic data integration capabilities of the XQuery language. The client query may itself inspect the time stamp values of the set of tuples. It may then decide itself to what extent a tuple is considered interesting yet stale. If the query decides that a given tuple is stale (e.g. if `type="networkLoad" AND TC < currentTime() - 10`), it calls the XQuery `document(URL contentLink)` function with the corresponding content link in order to pull and get handed fresh content, which it then processes in any desired way.

This mechanism makes it unnecessary for a hyper registry to guess what a client's notion of freshness might be. It also implies that a hyper registry does not require complex logic for query parsing, analysis, splitting, merging, etc. Moreover, the fresh results pulled by a query can be reused for subsequent queries. Since the query is executed within the hyper registry, the hyper registry may implement the `document` function such that it not only pulls and returns the current content, but as a side effect also updates the tuple cache in its database. A hyper registry retains its authority in the sense that it may apply an authorization policy, or perhaps ignore the query's refresh calls altogether and return the old content instead. The refresh-on-client-demand strategy is simple, elegant and controlled. It improves efficiency by avoiding overly eager refreshes typically incurred by a guessing hyper registry policy.

This basic idea could be used in variations that are more ambitious. For example, in an attempt to improve efficiency via "batching", a query may collect the content links of all tuples it considers stale into a set, and hand the set to a `documents(URL[])` function provided by the hyper registry, which then fetches fresh content in a batched fashion. Alternatively, a query may use this approach in a non-blocking manner, merely indicating that the hyper registry should soon refresh the given tuples (asynchronously), while the old tuples are still fine for the current query. The theme can be developed further. In a hierarchical P2P environment with caching nodes along the query route, it may be preferable to have the `documents(URL[])` function forward the refresh set as a query to the next node along the query route, instead of directly pulling from the content provider. This refreshes all node caches along the route, possibly at the expense of increased latency. As a different type of optimization, the hyper registry may reduce latency by keeping alive the TCP connections to content providers, which is often impractical to do for clients.

To summarize, a wide range of dynamic content freshness policies can be supported, which may be driven by all three system components: content provider, hyper registry and client. All three components may indicate how to manage content according to their respective notions of freshness.

## 4.8 Throttling

Clearly there is a tradeoff between the resource consumption caused by refreshes and state consistency. The higher the refresh frequency, the more consistent and up-to-date the state, and the more resources are consumed. High frequency refresh can consume significant network bandwidth, due to pathological client misbehavior, denial-of-service attacks, or sheer popularity. Implementations using high frequency refresh rates can encounter serious latency limitations due to the very expensive nature of secure (and even insecure) network connection

setup for publication. Keep-alive connections should be used to minimize setup time. However, they only partly address the problem. Consider, for example, an automatically adapting search engine indexing one million services, each refreshing every minute with a message of 200 bytes. Just to stay up-to-date the search engine must scale to 17000 refreshes/sec and its maintainer must pay for a WAN bandwidth of at least 3.4 MB/sec.

To condition for overload, limit resource consumption and satisfy minimum requirements on content freshness, mechanisms to *throttle* refresh frequency are proposed, adaptively inviting more or less traffic over time. For example, the search engine hyper registry may ask the content providers to wait at least 100 minutes between refreshes. Conversely, a hyper registry of a job scheduler depending on very fresh CPU load measurements from job execution services may want to invite execution service providers to refresh at least every second. For example, the service administrator can set the aggregate bandwidth available for refresh to a fraction of the total available system bandwidth. [74, 75] suggest to determine the maximum refresh rate for a given source service from historic bandwidth statistics over refreshes. This avoids overload, yet helps to maintain scalability in the number of source services.

Accordingly, the publication operation returns two time stamps `TS4` and `TS5`, which we call *minimum idle time* and *maximum idle time*, respectively. The semantics of the minimum idle time are as follows: *"Publication was successful, but in the future you may be dropped and denied service if you do not wait at least until time* `TS4` *before the next refresh"*. The semantics of the maximum idle time are as follows: *"Publication was successful, but in the future you may be dropped and denied service if you do not refresh before time* `TS5` *"*. We have `minimum idle time < maximum idle time`. A simple hyper registry always returns zero and infinity as minimum idle and maximum idle time, respectively. Content providers ignoring throttling warnings can be dropped and denied service without further notice, for example at the local, firewall or Internet Service Provider (ISP) level. Analogously, query operations return a minimum idle time (`TS4`) as part of the result set. The semantics are as follows: *"The query was successful, and here is the result set. However, in the future you may be dropped and denied service if you do not wait at least until time* `TS4` *before the next query"*.

## 4.9 Related Work

**Web Proxy Caches.** A weak cache coherency policy popular with web proxy caches is *adaptive TTL* [71]. Here the problem is handled by adjusting the time-to-live of a content based on observations of its lifetime. Adaptive TTL takes advantage of the fact that content lifetime distribution tends to be bimodal; if a given content has not been modified for a long time, it tends to stay unchanged. Thus, the time-to-live attribute of a given content is assigned to be a percentage of the content's current "age", which is the current time minus the last modified time of the document.

The "web server accelerator" [76] resides in front of one or more web servers to speed up user accesses. It provides an API, which allows application programs to explicitly add, delete, and update cached data. The API allows the accelerator to cache dynamic as well as static data. Invalidating and updating cached data is facilitated by the Data Update Propagation

(DUP) algorithm, which maintains data dependence information between cached data and underlying data in a graph [77].

**RDBMS.** Relational database systems provide SQL as a powerful query language. They do not support an XML data model and the XQuery language (see Section 3.6). Further, they do not provide soft state based publication, content caching and throttling. Content freshness is not addressed. Our work does not compete with an RDBMS, though. A hyper registry may well internally use an RDBMS for data management.

**UDDI.** UDDI (Universal Description, Discovery and Integration) [10] is an emerging industry standard that defines a business oriented access mechanism to a registry holding XML based WSDL service descriptions. It is not designed to be a registry holding arbitrary content. UDDI is not based on soft state, which implies that there is no way to dynamically manage and remove service descriptions from a large number of autonomous third parties in a reliable, predictable and simple way. It does not address the fact that services often fail or misbehave or are reconfigured, leaving a registry in an inconsistent state. Content freshness is not addressed. As such, UDDI only appears to be useful for businesses and their customers running static high availability services. Last, and perhaps most importantly, query support is rudimentary. Only key lookups with primitive qualifiers are supported, which is insufficient for realistic service discovery use cases (see Sections 3.4 and 3.5 for examples).

**Jini.** A Java client program begins discovery with a UDP multicast to locate instances of the Jini Lookup Service [63]. The network protocol is not language independent because it relies on the Java-specific object serialization mechanism. Publication is based on soft state. Clients and services must renew their leases periodically. Content freshness is not addressed. The query "language" allows for simple string matching on attributes, and is even less powerful than LDAP (see Section 3.6).

**SDS.** The Service Discovery Service (SDS) [26] is also based on multi cast and soft state. Content freshness is not addressed. It supports a simple XML based exact match query type. SDS is interesting in that it mandates secure channels with authentication and traffic encryption, and privacy and authenticity of service descriptions. SDS servers can be organized in a distributed hierarchy. For efficiency, each SDS node in a hierarchy can hold an index of the content of its sub-tree. The index is a compact aggregation and custom tailored to the narrow type of query SDS can answer.

**X500.** X500 [13] is the big legacy brother of LDAP, sharing the same data model and query language (see below). It was designed at a time when Internet protocols were not yet ubiquitous. X.500 requires ISO protocols, heavyweight ASN.1 data encoding and is much more complex than LDAP.

**LDAP.** The Lightweight Directory Access Protocol (LDAP) [14] defines an access mechanism in which clients send requests to and receive responses from LDAP servers. The

database is not based on soft state. Content freshness is not addressed. LDAP does not follow an XML data model. The expressive power of the LDAP query language is insufficient for service discovery use cases (see Sections 3.4 and 3.6) and most other non-trivial questions.

**MDS.** The Metacomputing Directory Service (MDS) [15, 16] is based on LDAP. As a result, its query language is insufficient for service discovery, and it does not follow an XML data model. MDS is based on soft state but it does not allow clients (and to some extent even content providers) to drive registry freshness policies.

The MDS consists of an unmodified OpenLDAP [78] server with value-adding backends, configured with a strong security library. OpenLDAP is an extensible open source LDAP server framework into which custom backend modules can be plugged, to which LDAP requests are dispatched. In terms of infrastructure, it could be seen as an early predecessor of Java servlet container technology.

The Grid Resource Information Service (GRIS) is an OpenLDAP backend that accepts LDAP queries from clients over its own LDAP namespace sub-tree. It is a backend into which a list of content providers can be plugged on a per attribute basis[7]. An example content provider is a shell script or program that returns the operating system version. A provider returns results in LDIF format, which is a text-based format for LDAP import and export. A provider can also be a module that is linked into the GRIS. A provider owns a namespace sub tree of the GRIS and returns a set of LDAP entries within that namespace. Depending on the namespace specified in an LDAP query, the GRIS executes (and creates the processes for) one or more affected providers and caches the results for use in future queries. It then applies the query against the cache. A GRIS can be statically configured to cache the pulled content for a fixed amount of time (on a per provider basis). An example GRIS configuration invokes the `/usr/local/bin/grid-info-cpu-linux` executable and caches CPU load results for 15 seconds.

Content provider invocation follows a CGI like life cycle model. The stateless nature, heavy weight process forking and context switches of such a model render it unsuitable for use in dynamic environments with high frequency refreshes and requests [79].

The Grid Index Information Service (GIIS) allows constructing and querying hierarchical LDAP directories by plugging together several GRIS (or GIIS) instances. A GRIS can publish a soft state description of itself in order to be added to the list of providers of a GIIS. The GIIS is nearly identical with the GRIS backend. However, it is configured to execute as content provider a function that forwards (chains) the query of the provider to other published LDAP server(s), which in most cases host a GRIS. GRIS and GIIS are identical with respect to query processing and caching. Multi-level directories are constructed by having a GIIS publish itself to another GIIS.

The publication process (mapped to the LDAP protocol's `add` request) is referred to as registration according to the Grid Registration Protocol (GRRP) protocol. The process of querying a server according to the LDAP protocol is referred to as "inquiry" according to the Grid Information Protocol (GRIP).

Figure 4.5 contrasts the different architectures of a GRIS and a hyper registry. A hyper

[7] A content provider is termed information providers in MDS parlance.

registry maintains content links and cached content in its database, whereas a GRIS maintains cached content only. The control paths from client to content provider and from content provider to the hyper registry are missing in the GRIS architecture, disabling cache freshness steering. A GRIS content provider is always local to the GRIS and cannot publish to a remote GRIS or GIIS. In contrast, a hyperlink content provider is cleanly decoupled from a hyper registry and only requires the ubiquitous HTTP protocol for simple communication with a local or remote hyper registry. A GRIS requires implementing the complex LDAP protocol, including its query language, at every content provider location. In contrast, handling the powerful but complex XQuery language is only required within a hyper registry, not at the content provider. The entry barrier to participation in our system is low. For example, it may be sufficient to run an off-the-shelf Apache server and `cron` job to publish content. Table 4.2 summarizes some commonalities and differences related to publication and content freshness.

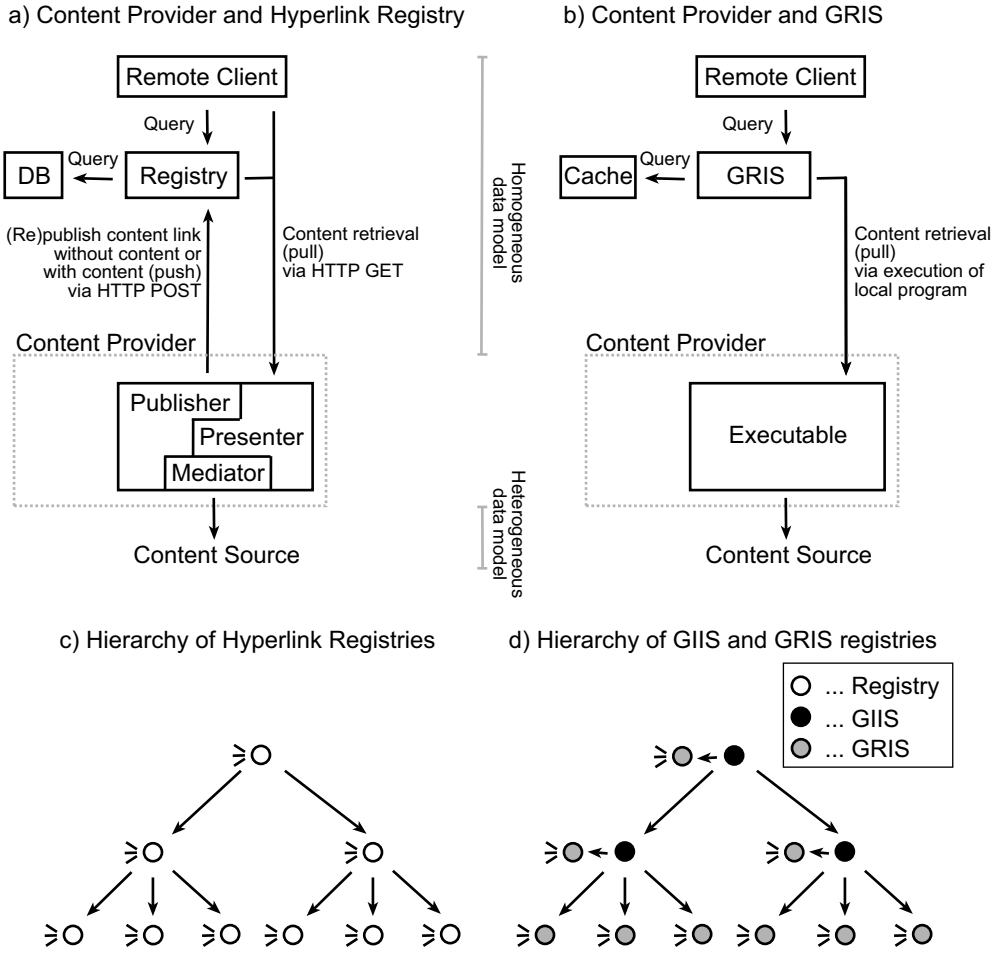


Figure 4.5: Hyper Registry and Grid Resource Information Service.

| Question | MDS | Hyper Registry |
|---|---|---|
| *Is the architecture uniform?* | No. There exist two concepts: GRIS and GIIS. | Yes. One concept is expressive enough – the hyper registry. |
| *Can a provider publish to a remote registry?* | No. A provider is local to a GRIS. | Yes. A provider can publish to any hyper registry, no matter whether the hyper registry is deployed locally, remotely or in-process. |
| *Can a provider actively steer registry freshness?* | No. A GRIS is actively pulling content. It can be statically configured to cache the pulled content for a fixed amount of time (on a per provider basis). A content provider is passive. It cannot actively publish and refresh content and hence cannot steer the freshness of its content cached in the GRIS. | Yes. Content provider and hyper registry are active and passive at the same time. At any time, a hyper registry can actively pull content, and a content provider can actively push with or without content. Both components can steer the freshness of content cached in the hyper registry. |
| *Can a client retrieve current content?* | No. A client cannot retrieve the current content from a content provider. It has to go through a GRIS or GIIS, which normally return stale content from their cache. | Yes. A client can directly connect to a content provider and retrieve the current content, thereby avoiding stale content from a hyper registry. |
| *Can a client query steer result freshness?* | No. A client query cannot steer the freshness of the results it generates. | Yes. A client query can steer the freshness of the results it generates via the refresh-on-client-demand strategy. |
| *Can a child registry publish content to a parent registry and steer its freshness?* | No. The GIIS is actively pulling content. A content provider and a GRIS are passive, cannot publish content to a remote registry (GIIS), and hence cannot actively steer the freshness of their respective content cached in a remote registry (GIIS). | No. A parent hyper registry is actively pulling content from its child hyper registries. A child hyper registry is passive, cannot publish content to a parent hyper registry, and hence cannot actively steer the freshness of content cached in a parent hyper registry. In this respect, MDS and hyper registry are alike. |
| *Can a parent registry cache content from a child registry, thereby trading content freshness for response time?* | Yes. A cache maintains results of prior queries. If the namespace of a query is contained in the namespace of the cache, the query is answered from the cache. | Maybe. We speculate that a hyper registry may use semantic caching [80] like the XCache system [81] to maintain prior queries and their result sets. The parts of a query overlapping with prior queries are answered from the cache. The remainder is a rewritten query that is evaluated like a normal (not cached) query. |

Table 4.2: Comparison of Hyper Registry and Metacomputing Directory Service.

## 4.10 Summary

**Comparison with Related Work.** We address the problems of maintaining dynamic and timely information populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources. The hyper registry has a number of key properties. An XML data model allows for structured and semi-structured data, which is important for integration of heterogeneous content. The XQuery language allows for powerful searching, which is critical for non-trivial applications. Database state maintenance is based on soft state, which enables reliable, predictable and simple content integration from a large number of autonomous distributed content providers. Content link, content cache and a hybrid pull/push communication model allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client.

These key properties distinguish our approach from related work, which individually addresses some, but not all of the above issues. Some work does not follow an XML data model (X.500, LDAP, MDS, RDBMS, JINI). Sometimes the query language is not powerful enough (UDDI, X.500, LDAP, MDS, JINI, SDS). Sometimes the database is not based on soft state (RDBMS, UDDI, X.500, LDAP). Sometimes content freshness is not addressed (RDBMS, UDDI, X.500, LDAP, JINI, SDS) or only partly addressed (MDS).

**Summary.** This chapter proposes a general-purpose XQuery database with mechanisms for content caching and soft state maintenance, the so-called *hyper registry*. The hyperlink hyper registry can be used to maintain hyperlinks and to cache content pointed to by these links. A content provider can publish a hyperlink, which in turn enables the hyper registry (and third parties) to pull (retrieve) the current content. Optionally, a content provider can also include a copy of the current content as part of publication. Example content includes service descriptions, XML documents or arbitrary binary content. A hyper registry may support caching, for example in server pull or client push mode or both. For reliable, predictable and simple distributed state maintenance, hyper registry tuples are maintained as *soft state*. A tuple may eventually be discarded unless refreshed by a stream of timely confirmation notifications from the provider. To condition for overload, limit resource consumption and satisfy minimum requirements on content freshness, mechanisms to *throttle* refresh and query frequency are proposed, adaptively inviting more or less traffic over time. Content link, content cache, a hybrid pull/push communication model and the expressive power of XQuery allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client. The entry barrier to participation in the system should be very low. To this end, the system is designed to be as simple as possible at the edges of the network. A content provider is cleanly decoupled from the hyper registry and only requires the ubiquitous HTTP protocol for communication with a local or remote hyper registry.

# Chapter 5

# The Web Service Discovery Architecture

## 5.1 Introduction

Having defined all registry aspects in detail in the previous chapter, we can now proceed to the definition of a web service layer that promotes interoperability for existing and future Internet software. Such a layer views the Internet as a large set of services with an extensible set of well-defined interfaces. A web service consists of a set of interfaces with associated operations. Each operation may be bound to one or more network protocols and endpoints. The definition of interfaces, operations and bindings to network protocols and endpoints is given as a service description [8]. In contrast to popular belief, a web service is neither required to carry XML [11] messages, nor to be bound to SOAP [9] or the HTTP [34] protocol, nor to run within a .NET [82] hosting environment. A discovery architecture defines appropriate services, interfaces, operations and protocol bindings for discovery. The key problem is:

- *Can we define a discovery architecture that promotes interoperability, embraces industry standards, and is open, modular, flexible, unified, non-intrusive and simple yet powerful?*

We propose and specify such a discovery architecture, the so-called *Web Service Discovery Architecture (WSDA)*. WSDA subsumes an array of disparate concepts, interfaces and protocols under a single semi-transparent umbrella. It specifies a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery. These primitives cover service identification, service description retrieval, data publication as well as minimal and powerful query support. The individual primitives can be combined and plugged together by specific clients and services to yield a wide range of behaviors and emerging synergies. We define four interfaces, namely `Presenter`, `Consumer`, `MinQuery` and `XQuery`. The `Presenter` interface allows clients to retrieve the current service description. The `Consumer` interface allows content providers to publish a tuple set to a consumer. The `MinQuery` interface provides the simplest possible query support ( *"select all"*-style); It returns tuples including or excluding cached content. The `XQuery` interfaces provides powerful XQuery support. Finally, we compare in detail the properties of WSDA with the emerging Open Grid Service Architecture [6, 17].

## 5.2 Interfaces

**Presenter.** The `Presenter` interface allows clients to retrieve the current service description. Clearly clients from anywhere must be able to retrieve the current description of a service (subject to security policy). Hence, a service needs to present (make available) to clients the means to retrieve the service description. To enable clients to query in a global context, some identifier for the service is needed. Further, a description retrieval mechanism is required to be associated with each such identifier. Together these are the bootstrap key (or handle) to all capabilities of a service. In principle, identifier and retrieval mechanisms could follow any reasonable convention.

In practice, however, a fundamental mechanism such as service discovery can only hope to enjoy broad acceptance, adoption and subsequent ubiquity if integration of legacy services is made easy. The introduction of service discovery as a new and additional auxiliary service capability should require as little change as possible to the large base of valuable existing legacy services, preferable no change at all. It should be possible to implement discovery-related functionality without changing the core service. Further, to help easy implementation the retrieval mechanism should have a very narrow interface and be as simple as possible.

In support of these requirements, the identifier is chosen to be a URL [65], and the retrieval mechanism is chosen to be HTTP(S) [34]. We define that an HTTP(S) GET request to the identifier must return the current service description (subject to local security policy). In other words, a simple hyperlink is employed. In the remainder of this thesis, we will use the term *service link* for such an HTTP URL identifier enabling service description retrieval. Like in the WWW, service links (and content links, see below) can freely be chosen as long as they conform to the HTTP URL specification [65]. Hence, they may contain the usual URL encoded attribute-value pairs. The semantics of the structure of a given link are opaque to a client. Examples for legal links are:

```
http://sched.cern.ch:8080/getServiceDescription.wsdl
https://cms.cern.ch/getServiceDescription?id=4712&cache=disable
http://phone.cern.ch/lookup?query="select phone from phonebook where phone=0450-1234"
http://repcat.cern.ch/getPhysicalFileNames?lfn="myLogicalFileName"
```

Because service descriptions should describe the essentials of the service, it is recommended[1] that the service link concept be an integral part of the description itself. As a result, service descriptions may be retrievable via the `Presenter` interface, which defines an operation `getServiceDescription()` for this purpose. The operation is identical to service description retrieval and is hence bound to (invoked via) an HTTP(S) GET request to a given service link. Additional protocol bindings may be defined as necessary.

**Consumer.** The `Consumer` interface allows content providers to publish a tuple set to a consumer. A WSDA *tuple* follows the Dynamic Data Model (DDM) (see Section 3.3). It has as attributes a content link, a type, a context, four soft state time stamps, and (optionally)

[1] In general, it is not mandatory for a service to implement any "standard" interface. Historical evidence suggests that the acceptance of ubiquituous Internet infrastructures and their flexible and successful evolution strongly depends on being conservative with the term *MUST*.

two arbitrary-shaped extensibility elements, namely metadata and content. A content link (e.g. service link) is an HTTP(S) URL that may point to the content of a content provider. Given the link the current content can be retrieved (pulled) at any time via an HTTP(S) GET request to the link. The type describes *what* kind of content is being published. The context describes *why* the content is being published or *how* it should be used. The optional metadata element may further describe the content and/or its retrieval beyond what can be expressed with the previous attributes. For example it may describe retrieval from an UDDI [10] registry, formulated in the Web Service Inspection Language (WSIL) [69], or it may be a secure digital XML signature [68]. Given this tuple information, a content *retriever* module can retrieve the current content from the provider. Based on embedded soft state time stamps, a tuple may eventually be discarded unless refreshed by a stream of timely confirmation notifications. Within a tuple set, a tuple is uniquely identified by its *tuple key*, which is the pair `(content link, context)`. Content and metadata can be structured or semi-structured data in the form of any arbitrary well-formed XML document or fragment[2]. An individual element may, but need not, have a schema (XML Schema [12]), in which case it must be valid according to the schema. All elements may, but need not, share a common schema. This flexibility is important for integration of heterogeneous content. To summarize, a WSDA tuple is an annotated multi-purpose soft state data container that may contain a piece of arbitrary content and allows for refresh of that content at any time, as depicted in Figure 5.1. WSDA offers an open dynamic data model that allows for a wide range of powerful caching policies. The publish operation has the signature `(TS4, TS5) publish(XML tupleset)`. For detailed motivation, justification and discussion of the Dynamic Data Model and the semantics of soft state time stamps, see Chapters 3 and 4.

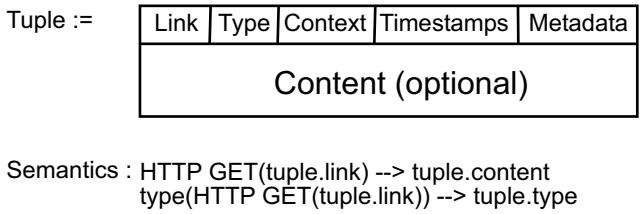


Figure 5.1: Tuple Link allows for Refresh of Tuple Content at any time.

**MinQuery.** The `MinQuery` interface provides the simplest possible query support ( *"select all"*-style). It returns tuples including or excluding cached content. As a minimum, clients can query by invoking minimalist query operations such as `getTuples()`. The `getTuples()` query operation takes no arguments and returns the full set of all tuples "as is". That is, query output format and publication input format are the same. If supported, output includes cached content. The `getLinks()` query operation is similar in that it also takes no

[2]For clarity of exposition, the content is an XML element. In the general case (allowing non-text based content types such as `image/jpeg`), the content is a MIME [67] object. The XML based publication input tuple set and query result tuple set is augmented with an additional MIME multipart object, which is a list containing all content. The content element of a tuple is interpreted as an index into the MIME multipart object.

arguments and returns the full set of all tuples. However, it always substitutes an empty string for cached content. In other words, the content is omitted from tuples, potentially saving substantial bandwidth. For an extensive discussion, see Chapter 4.

Advanced query support can be expressed on top of the minimal query capabilities. Such higher-level capabilities conceptually do not belong to a consumer and minimal query interface, which are only concerned with the fundamental capability of making a content link (e.g. service link) *reachable*[3] for clients. As an analogy, consider the related but distinct concepts of web hyper-linking and web searching: Web hyper-linking is a fundamental capability without which nothing else on the Web works. Many different kinds of web search engines using a variety of search interfaces and strategies can and are layered on top of web linking. The kind of XQuery support we propose below is certainly not the only possible and useful one. It seems unreasonable to assume that a single global standard query mechanism can satisfy all present and future needs of a wide range of communities. Multiple such mechanisms should be able to coexist. Consequently, the consumer and query interfaces are deliberately separated and kept as minimal as possible, and an additional interface type (`XQuery`) for answering XQueries is introduced below.

**XQuery.** The `XQuery` interface provides powerful XQuery support, which is important for realistic service and resource discovery use cases (see Section 3.5). For a detailed motivation and justification, including a discussion of a wide range of discovery queries and an evaluation of various query languages, see Chapter 3. XQuery [18, 50, 51, 52] is the standard XML query language developed under the auspices of the W3C. It allows for powerful searching, which is critical for non-trivial applications. Everything that can be expressed with SQL [19] can also be expressed with XQuery. However, XQuery is a more expressive language than SQL. XQuery can dynamically integrate external data sources via the `document(URL)` function. The `document(URL)` function can be used to process the XML results of remote operations invoked over HTTP. For example, given a service description with a `getPhysicalFileNames(LogicalFileName)` operation, a query can match on values dynamically produced by that operation. The same rules that apply to minimalist queries also apply to XQuery support. An implementation can use a modular and simple XQuery processor such as `Quip` [70] for the operation `XML query(XQuery query)`. Because not only content, but also content link, context, type, time stamps, metadata etc. are part of a tuple, a query can also select on this information.

**Interface Summary.** The four interfaces and their respective operations are summarized in Table 5.1. Figure 5.2 depicts the interactions of a client with implementations of these interfaces.

[3] *Reachability* is interpreted in the spirit of garbage collection systems: A content link is reachable for a given client if there exists a direct or indirect retrieval path from the client to the content link.

| Interface | Operations | Responsibility |
|---|---|---|
| `Presenter` | `XML getServiceDescription()` | Allows clients to retrieve the current description of a service and hence to bootstrap all capabilities of a service. See Section 2.3. |
| `Consumer` | `(TS4,TS5) publish(XML tupleset)` | A content provider can publish a dynamic pointer called a content link, which in turn enables the consumer (e.g. hyper registry) to retrieve the current content. Optionally, a content provider can also include a copy of the current content as part of publication. Each input tuple has a content link, a type, a context, some time stamps, and (optionally) metadata and content. See Section 4.2 and 2.4. |
| `MinQuery` | `XML getTuples()`<br>`XML getLinks()` | Provides the simplest possible minimal query support (*"select all"*-style). The `getTuples` query operation returns the full set of all available tuples "as is". The `getLinks` query operation is identical, except that it always substitutes an empty string for cached content. See Section 4.4 and 2.4. |
| `XQuery` | `XML query(XQuery query)` | Provides powerful XQuery support. Executes an XQuery over the available tuple set. Because not only content, but also content link, context, type, time stamps, metadata etc. are part of a tuple, a query can also select on this information. See Section 4.4 and 3.5. |

Table 5.1: WSDA Interfaces and their Respective Operations.

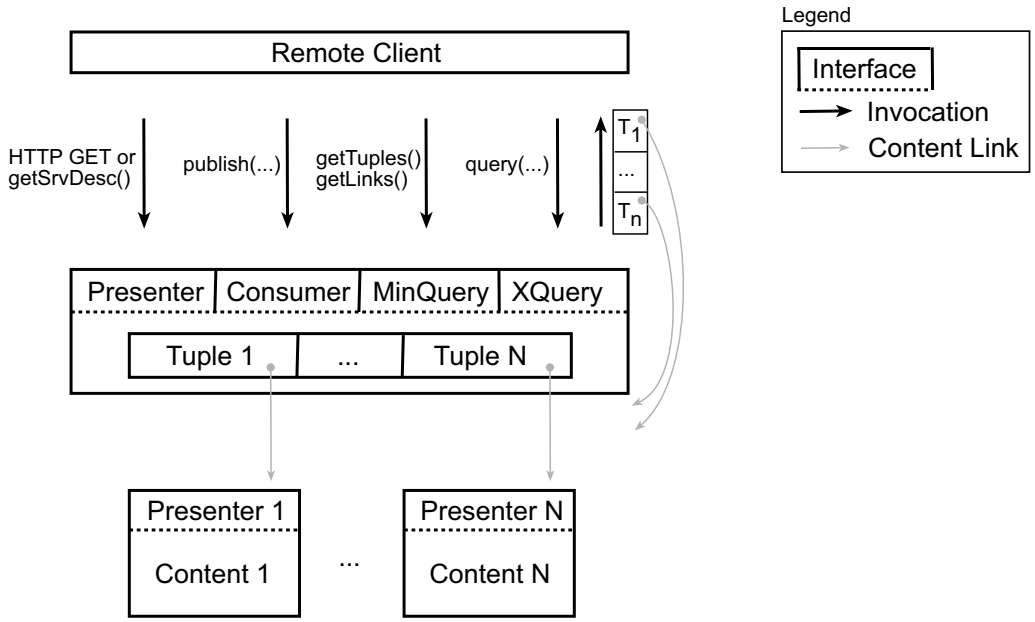


Figure 5.2: Interactions of Client with Implementations of WSDA Interfaces.

## 5.3 Network Protocol Bindings

The operations of the WSDA interfaces are bound to (carried over) a default transport protocol. The `XQuery` interface is bound to the *Peer Database Protocol (PDP)* proposed in Chapter 7. For all other operations and arguments we assume for simplicity HTTP(S) GET and POST as transport, and XML based parameters. *Additional* protocol bindings may be defined as necessary. An example service description of a registry implementing all four WSDA interfaces, formulated in SWSDL, is depicted in Figure 5.3.

```
<service>
  <interface type ="http://gridforum.org/interface/Presenter-1.0">
    <operation>
      <name>XML getServiceDescription()</name>
      <bind:http verb="GET" URL="https://registry.cern.ch/getServiceDescription"/>
    </operation>
  </interface>

  <interface type = "http://gridforum.org/interface/Consumer-1.0">
    <operation>
      <name> (Time TS4, Time TS5) publish(XML tupleset)</name>
      <bind:http verb="POST" URL="https://registry.cern.ch/publish"/>
    </operation>
  </interface>

  <interface type = "http://gridforum.org/interface/MinQuery-1.0">
    <operation>
      <name> XML getTuples()</name>
      <bind:http verb="GET" URL="https://registry.cern.ch/getTuples"/>
    </operation>
    <operation>
      <name> XML getLinks()</name>
      <bind:http verb="GET" URL="https://registry.cern.ch/getLinks"/>
    </operation>
  </interface>

  <interface type = "http://gridforum.org/interface/XQuery-1.0">
    <operation>
      <name> XML query(XQuery query)</name>
      <bind:beep URL="beep://registry.cern.ch:9000"/>
    </operation>
  </interface>
</service>
```

Figure 5.3: SWSDL description of a registry service implementing all four WSDA interfaces.

## 5.4 Services

In Chapter 4 we defined two kinds of example registry services: The so-called *hypermin registry* must (at least) support the three interfaces `Presenter`, `Consumer` and `MinQuery` (excluding XQuery support). A *hyper registry* must (at least) support these interfaces plus the `XQuery` interface. Put another way, any service that happens to support, among others, the respective interfaces qualifies as a hypermin registry or hyper registry. As usual, the interfaces may have endpoints that are hosted by a single container, or they may be spread across multiple hosts or administrative domains.

It is by no means a requirement that only dedicated hyper registry services and hypermin registry services may implement WSDA interfaces. Any arbitrary service may decide to offer and implement none, some or all of these four interfaces. For example, a job scheduler may decide to implement, among others, the `MinQuery` interface to indicate a simple means to discover metadata tuples related to the current status of job queues and the supported Quality of Service. The scheduler may not want to implement the `Consumer` interface because its metadata tuples are strictly read-only. Further, it may not want to implement the `XQuery` interface, because it is considered overkill for its purposes. Even though such a scheduler service does not qualify as a hypermin or hyper registry, it clearly offers useful added value. Other examples for services implementing a subset of WSDA interfaces are consumers such as an instant news service or a cluster monitor. These services may decide to implement the `Consumer` interface to invite external sources for data feeding, but they may not find it useful to offer and implement any query interface.

In a more sophisticated scenario, the example job scheduler may decide to publish its local tuple set also to an (already existing) remote helper hyper registry service (i.e. with XQuery support). To indicate to clients how to get hold of the XQuery capability, the scheduler may simply copy the `XQuery` interface description of the remote helper hyper registry service and advertise it as its own interface by including it in its own service description. This kind of *virtualization* is not a "trick", but a feature with significant practical value, because it allows for minimal implementation and maintenance effort on the part of the scheduler.

Alternatively, the scheduler may include in its local tuple set (obtainable via the `getLinks()` operation) a tuple that refers to the service description of the remote helper hyper registry service. An interface referral value for the context attribute of the tuple is used, as follows:

```
<tuple link="https://registry.cern.ch/getServiceDescription"
       type="service" ctx="x-ireferral://gridforum.org/interface/XQuery-1.0"
       TS1="30" TC="0" TS2="40" TS3="50">
</tuple>
```

## 5.5 Properties

WSDA has a number of key properties:

- **Standards Integration.** WSDA embraces and integrates solid and broadly accepted industry standards such as XML [11], XML Schema [12], the Simple Object Access Protocol (SOAP) [9], the Web Service Description Language (WSDL) [8] and XQuery

[18]. It allows for integration of emerging standards such as the Web Service Inspection Language (WSIL) [69].

- **Interoperability.** WSDA promotes an interoperable web service layer on top of existing and future Internet software, because it defines appropriate services, interfaces, operations and protocol bindings. WSDA does not introduce new Internet standards. Rather, it judiciously combines existing interoperability-proven open Internet standards such as HTTP(S) [34], URI [65], MIME [67], XML [11], XML Schema [12] and BEEP [35, 36, 31].

- **Modularity.** WSDA is modular because it defines a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery. These primitives cover service identification, service description retrieval, publication, as well as minimal and powerful query support. The responsibility, definition and evolution of any given primitive is distinct and independent of that of all other primitives.

- **Ease-of-use and Ease-of-implementation.** Each communication primitive is deliberately designed to avoid any unnecessary complexity. The design principle is to *"make simple and common things easy, and powerful things possible"*. In other words, solutions are rejected that provision for powerful capabilities yet imply that even simple problems are complicated to solve. For example, service description retrieval is by default based on a simple HTTP(S) GET. Yet, we do not exclude, and indeed allow for, alternative identification and retrieval mechanisms such as the ones offered by UDDI (Universal Description, Discovery and Integration) [10], RDBMS or custom Java RMI registries (e.g. via tuple metadata specified in WSIL [69]). Further, tuple content is by default given in XML, but advanced usage of arbitrary MIME [67] content (e.g. binary images, files, MS-Word documents) is also possible. As another example, the minimal query interface requires virtually no implementation effort on the part of a client or server. Yet, where necessary, also powerful XQuery support may, but need not, be implemented and used.

- **Openness and Flexibility.** WSDA is open and flexible because each primitive can be used, implemented, customized and extended in many ways. For example, the interfaces of a service may have endpoints spread across multiple hosts or administrative domains. However, there is nothing that prevents all interfaces to be co-located on the same host or implemented by a single program. Indeed, this is often a natural deployment scenario. Further, even though default network protocol bindings are given, additional bindings may be defined as necessary. For example, an implementation of the `Consumer` interface may bind to (carry traffic over) HTTP(S) [34], SOAP/BEEP [31], FTP [32], or RMI [83]. The tuple set returned by a query may be maintained according to a wide variety of cache coherency policies, resulting in static to highly dynamic behavior. A consumer may take any arbitrary custom action upon publication of a tuple. For example, it may interpret a tuple from a specific schema as a command or an active message [84], triggering tuple transformation and/or forwarding to other consumers such as loggers. For flexibility, a service maintaining a WSDA tuple set may be deployed in any arbitrary

way. For example, the database can be kept in a XML file, in the same format as returned by the `getTuples` query operation. However, tuples can also be kept in a remote relational database.

- **Expressive Power.** WSDA is powerful because its individual primitives can be combined and plugged together by specific clients and services to yield a wide range of behaviors. Each single primitive is of limited value all by itself. The true value of simple orthogonal multi-purpose communication primitives lies in their potential to generate powerful emerging synergies. For example, combination of WSDA primitives enables building services for data management, workflow management, auditing, instrumentation, monitoring and problem determination.

  As another example, the consumer and query interfaces can be combined to implement a Peer-to-Peer (P2P) database network for service discovery. In a large distributed system spanning many administrative domains such as a DataGrid, it is desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. However, in such a database system, the set of information tuples in the universe is partitioned over one or more distributed nodes, for reasons including autonomy, scalability, availability, performance and security. Here, P2P nodes maintain a local database and implement the consumer and query interfaces. Clients and P2P nodes publish (their) service descriptions and/or other metadata to one or more P2P nodes. Publication enables distributed node topology construction (e.g. ring, tree or graph) and at the same time constructs the database to be searched. When any client wishes to search the P2P network with some query (see, for example Section 3.5), it sends the query to a single node. The node applies the query to its local database and returns matching results; it also forwards the query to its neighbor nodes. These neighbors return their local query results; they also forward the query to their neighbors, and so on. For an extensive discussion, see Chapter 6 and Chapter 7, where a P2P database framework and corresponding network protocol are devised, which are unified in the sense that they allow to express specific applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options.

- **Uniformity.** WSDA is unified because it subsumes an array of disparate concepts, interfaces and protocols under a single *semi-transparent* umbrella. It allows for multiple competing distributed systems concepts and implementations to coexist and to be integrated. Clients can dynamically adapt their behavior based on rich service introspection capabilities. Clearly there exists no solution that is optimal in the presence of the heterogeneity found in real-world large cross-organizational distributed systems such as Data Grids, electronic market places and instant Internet news and messaging services. Introspection and adaption capabilities increasingly make it unnecessary to mandate a single global solution to a given problem, thereby enabling integration of collaborative systems.

- **Non-Intrusiveness.** WSDA is non-intrusive because it offers interfaces but does not mandate that every service in the universe must comply to a set of "standard"

interfaces.

## 5.6 Comparison with Open Grid Services Architecture

We have recently learned about the emerging *Open Grid Services Architecture (OGSA)* [6, 17]. OGSA exhibits striking similarities with the Web Service Discovery Architecture (WSDA) proposed above, in spirit and partly also in design. We stress that this thesis and OGSA have so far been mutually independent work in their entirety. Future work is likely to be collaborative and convergent due to shared interest. Due to the recent circulation of early OGSA material, our understanding of it is limited and not necessarily accurate. Nevertheless, in this section we attempt a preliminary comparison of OGSA concepts with WSDA concepts.

OGSA is work-in-progress, but an important first step towards enabling powerful, flexible yet also interoperable large cross-organizational Grid systems. Like WSDA, OGSA defines and standardizes a set of (mostly) orthogonal multi-purpose communication primitives that can be combined and customized by specific clients and services to yield powerful behavior. Like WSDA, OGSA embraces solid and broadly accepted industry standards such as XML [11], XML Schema [12], the Simple Object Access Protocol (SOAP) [9], the Web Service Description Language (WSDL) [8] and XQuery [18].

**Service Link, Service Description and Presenter.** In OGSA, a service instance is identified by a *Grid Service Handle (GSH)*, which is an immutable, globally unique HTTP(S) URL that distinguishes a specific service instance from all other service instances that have existed, exist now, or will exist in the future. By means of an HTTP GET or the `HandleMap` interface, the handle can be resolved to a *Grid Service Reference (GSR)*, which is typically a WSDL document containing descriptions of all supported service interfaces. A reference may change over time. That is, the contents of the WSDL document may change over time. A reference is based on soft state, hence expires unless periodically renewed.

A GSH corresponds to a WSDA *service link*. However, unlike a GSH, a service link is neither required to be immutable nor globally unique. A GSR corresponds to a WSDA *service description* given in WSDL. Both are soft state documents. Although it is unclear from the complex presentation, it appears that a `HandleMap` corresponds to the WSDA `Presenter` interface. In WSDA, the operation `Presenter.getServiceDescription()` or a simple HTTP(S) GET to a content link (e.g. service link) may be used to retrieve the current content (e.g. service description). WSDA allows arbitrary-shaped content retrieval via MIME encoding (e.g. textual, binary), including mapping a service link to a service description. It appears that OGSA is restricted to mapping a GSH to a GSR. Further not every legal HTTP(S) URL is a legal GSH. In WSDA, every legal HTTP(S) URL is a legal content link and hence also a legal service link. OGSA defines implicit and unclear URL suffix mapping and `GSHomeHandleMapID` semantics that we believe would better be omitted or expressed as part of the `HandleMap` interface. The OGSA `HandleMap` operation `GSR FindByHandle(GSH)` corresponds to the WSDA `Presenter` operation `XML getServiceDescription()`. The additional GSH argument is unnecessary in our approach.

**Tuple, Tuple Set and Query.** In OGSA, a *grid service instance* maintains so-called *service data* XML elements. A service data element is a multi-purpose soft state data container that may contain arbitrary content. A service data element has a name attribute and three soft state time stamp attributes (`goodFrom`, `goodUntil`, `notGoodAfter`) and may contain an arbitrary extensibility element as content. In contrast, a WSDA *tuple* follows the Dynamic Data Model (DDM). It has as attributes a content link, a type, a context, four soft state time stamps, and (optionally) two arbitrary-shaped extensibility elements, namely metadata and content. A WSDA tuple is an annotated multi-purpose soft state data container that may contain a piece of arbitrary content and allows for refresh of that content at any time (see Figure 5.1). A collection of OGSA service data elements corresponds to a WSDA *tuple set*. In WSDA, publication input data and query output is uniformly expressed as a tuple set[4].

The OGSA operation `FindServiceData` of the `GridService` interface allows querying the collection of service data elements (roughly corresponding to the WSDA `MinQuery` and `XQuery` interfaces). It attempts to support multiple query languages by accepting and returning an arbitrary XML element. The schema of the input XML element indicates the query language. If the service instance supports the desired query language, the query is executed against the collection of service data elements. Every OGSA grid service must support a simple query "language" that returns a list of all service data elements whose name equals a given name (exact match). The name *"root"* is reserved and must return at least a list of "standard" service data elements such as handle, reference, primary key, a list of the supported query languages, etc.

The OGSA `FindServiceData` operation takes arbitrary XML input and returns arbitrary XML output. It remains to be seen how useful it is to coerce distinct query capabilities into a single generic operation. Consider that modern software systems rarely coerce distinct capabilities into a single generic handler function of the form `Object do(Object)`. Further consider that the very purpose of separate interface types and names is to allow for independence, separate evolution, flexibility, clarity, predictability, type safety and straightforward introspection. In essence, this is what web services and service descriptions are about. In this light, generic functions for distinct capabilities appear counter-intuitive to the spirit of web services.

For comparison, in WSDA, trivial and powerful query support are cleanly separated. The `MinQuery` interface is indeed minimal and requires only the simplest possible query support (*"select all"*-style) via the operations `getTuples()` and `getLinks()`. The WSDA `XQuery` interface, on the other hand, allows extremely powerful queries. Finally, it is unclear from the material whether OGSA intends in the future to support either or both XQuery, XPath, or none.

**Data Publication.** An OGSA *registry service instance* maintains a collection of handles. Typically, an OGSA registry service offers a `Registry` interface, which supports registering and unregistering handles. The `registerService` operation takes as input a handle, a timeout, and optionally, an abstract and an arbitrary extensibility element. The

[4]The output tuple set of a constructive query may contain arbitrary content, whereas all other queries output a tuple set with tuples from the dynamic data model (see Section 3.3).

`unregisterService` operation takes as input the handle to be removed.

In contrast, a WSDA consumer and query can accept and return arbitrary textual and binary soft state data in the form of a tuple set, including content links, service links (handles), cached content and metadata (e.g. a WSIL [69] fragment). The OGSA `registerService` operation roughly corresponds to the WSDA `publish` operation. However, the latter operation is a unified multi-purpose operation, supporting a set of arbitrary-shaped soft state tuples. The functionality to publish information other than handles (e.g. references) to a registry or service currently appears to be missing from OGSA. The OGSA notification interface appears only useful for interaction patterns based on subscription by notification sinks. Invitation for subscription appears to be missing. The `unregisterService` operation is not necessary in WSDA, because a tuple is based on soft state. Explicit (immediate) unregistration can be achieved by using the WSDA `publish` operation with tuples carrying zero-valued soft state time stamps.

An OGSA registry service supports discovery queries via the `FindServiceData` operation of the `GridService` interface. The relationship between discovery queries and the maintained collection of handles is unclear. More precisely, it is unclear whether handles are maintained as service data elements, and if they can be queried in the same way as described above.

The OGSA `NotificationSource` and `NotificationSink` interfaces allow for publish-subscribe functionality based on message type and interest statements. A notification source may offer a set of topics, which notification sinks may use for subscription. A notification source may send notification messages to a subscribed notification sink. Subscription requests are soft state based and expire unless periodically renewed. The OGSA `NotificationSink` interface corresponds to the WSDA `Consumer` interface. However, it appears that it is not foressen that an OGSA notification message may carry a set of more than one service data elements[5]. In contrast, a WSDA consumer message explicitly carries zero or more tuples in a tuple set, resulting in potentially much improved efficiency. Consider that all production quality database management systems we are aware of support batching of tuples over the network. This is because inserting a million tuples (e.g. CPU load samples) into a database should not involve the same number of network round-trips. In addition to efficiency by design, WSDA offers an open and precisely specified dynamic data model that allows for a wide range of powerful caching policies. We are working on a multi-purpose interface for persistent XQueries (i.e. server-side trigger queries), which will roughly correspond to the OGSA `NotificationSource` interface, albeit in a more general and powerful manner.

**Grid Service.** In OGSA, a `GridService` interface supports discovery queries (`findServiceData`), setting and prolonging of shutdown time (`setTerminationTime`) as well as explicit (immediate) shutdown of the service (`destroy`). OGSA mandates that every grid service must implement the `GridService` interface. In contrast, WSDA does not require a service to implement any "standard" interface. A specific service (e.g. the hyper registry service) may, of course, mandate implementation of certain interfaces, but there is no global requirement that any and all services in the universe must satisfy. Historical evidence suggests that the

[5]Nesting service data elements inside a service data element is possible but undefined and left without semantics.

acceptance of ubiquituous Internet infrastructures and their flexible and successful evolution strongly depends on being conservative with the term *MUST*. A design rarely turns out right the first time. Typically, several design revisions and refactorings over time are needed. Consider that once a fundamental interface is introduced as mandatory, one is "stuck" with it forever, at least if compatibility and stability are of concern.

The problem is subtle and comparable to the design of the base class of single-rooted object oriented programming languages. A uniform and well designed base class clearly offers strong advantages because everyone can safely assume certain essential features to be available everywhere. Many C++ reusability problems stem from the fact that the C++ language has no single common base class. For example, integration of third-party frameworks is problematic at best. On the other hand, a controversial or flawed base class feature such as the (potentially very useful) `clone()` method of the Java base class is bound to lead to dissatisfaction.

Another comparable problem-in-the-large is the definition of the Java platform. Typically, Java interfaces and frameworks are not designed, specified, revised, standardized and hardened *within* the Java platform. Rather, they are born and live externally for some two years to allow enough time for a community process, deployment feedback from reference implementations, and for separation of wheat from chaff. Only then may some of them be considered to be merged into the core Java platform definition.

It is often desirable to include features only if considered non-intrusive and absolutely essential for a wide range of communities. While certainly interesting and often useful, justification is missing why query, shutdown and other lifetime maintenance of services should be absolutely essential for *every* service. Further, while certainly well designed, justification is missing why the chosen definition of these features is superior to other approaches. Consider that a large variety of server administration and monitoring products with related but not equivalent interfaces has been introduced and marketed in the past [85]. In addition, the OGSA query interface certainly is, just like our query interfaces, not the only possible and useful one. Finally, we believe that a main idea behind web services is service description introspection and dynamic adaption. This capability increasingly makes it unnecessary to mandate a global "service base class or interface".

**Other.** The OGSA `Factory` interface supports dynamically creating short or long-lived remote service instances. An OGSA service instance may be associated with a *primary key*, which may be used to locate and shut down service instances created by a factory. It is unclear what the added value of a primary key over a handle is. The concept may perhaps be related to the WSDA *tuple key*, which is the pair `(content link, context)`. A tuple key uniquely identifies a tuple within a tuple set.

Table 5.2 summarizes the comparison of corresponding OGSA and WSDA concepts.

## 5.7 Summary

We propose and specify an open discovery architecture, the so-called *Web Service Discovery Architecture (WSDA)*. WSDA views the Internet as a large set of services with an extensible

| **Concept** | **OGSA** | **WSDA** |
|---|---|---|
| *Interfaces* | GridService, NotificationSource, NotificationSink, Registry, Factory, PrimaryKey, HandleMap | Presenter, Consumer, MinQuery, XQuery |
| *Interfaces required to be implemented by every service* | GridService interface; defines operations for query and life cycle maintaince | None |
| *Service identifier* | Grid Service Handle (GSH) | Service link (i.e. content link) |
| *Service description* | Grid Service Reference (GSR) (e.g. WSDL) | Service description (e.g. WSDL) |
| *Service description retrieval* | via HTTP(S) GET or `HandleMap.findByHandle(GSH)` | via HTTP(S) GET or `Presenter.getServiceDescription()` |
| *Multi-purpose data container* | Service data | Tuple |
| *Set of data containers* | Service data list | Tuple set |
| *Query capability* | `GridService.FindServiceData(XML query)` | `MinQuery.getLinks()`, `MinQuery.getTuples()`, `XQuery.query(XQuery)` |
| *Data publication* | `Registry.RegisterService(handle)`, `NotificationSink.deliverNotification (servicedata)` | `Consumer.publish(XML tupleset)` |

Table 5.2: Open Grid Service Architecture vs. Web Service Discovery Architecture.

set of well-defined interfaces. WSDA has a number of key properties. It promotes an interoperable web service layer on top of existing and future Internet software, because it defines appropriate services, interfaces, operations and protocol bindings. It embraces and integrates solid and broadly accepted industry standards such as XML, XML Schema, the Simple Object Access Protocol (SOAP), the Web Service Description Language (WSDL) and XQuery. It allows for integration of emerging standards such as the Web Service Inspection Language (WSIL). It is modular because it defines a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery. These primitives cover service identification, service description retrieval, data publication as well as minimal and powerful query support. Each communication primitive is deliberately designed to avoid any unnecessary complexity. WSDA is open and flexible because each primitive can be used, implemented, customized and extended in many ways. It is powerful because the individual primitives can be combined and plugged together by specific clients and services to yield a wide range of behaviors and emerging synergies. It is unified because it subsumes an array of disparate concepts, interfaces and protocols under a single semi-transparent umbrella. It is non-intrusive because it offers interfaces but does not mandate that every service in the universe must comply to a set of "standard" interfaces. Finally, we compare in detail the properties of WSDA with the emerging Open Grid Service Architecture.

Tim Berners-Lee designed the World Wide Web as a consistent interface to a flexible and changing heterogeneous information space for use by CERN's staff, the High Energy Physics community, and, of course, the world at large. The WWW architecture [86] rests on four simple and orthogonal pillars: URIs as identifiers, HTTP for retrieval of content pointed

to by identifiers, MIME for flexible content encoding, and HTML as the primus-inter-pares (MIME) content type. Based on our Dynamic Data Model (DDM), we hope to proceed further towards a self-describing meta content type that retains and wraps all four WWW pillars "as is", yet allows for flexible extensions in terms of identification, retrieval and caching of content. Judicious combination of the four WWW pillars, DDM, WSDA, the Hyper Registry, the Unified Peer-to-Peer Database Framework (UPDF) and its associated Peer Database Protocol (PDP) are used to define how to bootstrap, query and publish to a dynamic and heterogeneous information space maintained by self-describing network interfaces.

# Chapter 6

# A Unified Peer-to-Peer Database Framework

In a distributed system, it is desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. As in a data integration system, the goal is to exploit several independent information sources as if they were a single source. However, in a large distributed database system spanning many administrative domains, the set of information tuples in the universe is partitioned over one or more distributed nodes, for reasons including autonomy, scalability, availability, performance and security. It is not obvious how to enable powerful discovery query support and collective collaborative functionality that operate on the distributed system as a whole, rather than on a given part of it. Further, it is not obvious how to allow for search results that are fresh, allowing dynamic content. It appears that a Peer-to-Peer (P2P) database network may be well suited to support dynamic distributed database search, for example for service discovery.

The overall P2P idea is as follows. Rather than have a centralized database, a distributed framework is used where there exist one or more autonomous database nodes, each maintaining its own data. Queries are no longer posed to a central database; instead, they are recursively propagated over the network to some or all database nodes, and results are collected and send back to the client. The key problems then are:

- *What are the detailed architecture and design options for P2P database searching in the context of service discovery? What response models can be used to return matching query results? How should a P2P query processor be organized? What query types can be answered (efficiently) by a P2P network? What query types have the potential to immediately start piping in (early) results? How can a maximum of results be delivered reliably within the time frame desired by a user, even if a query type does not support pipelining? How can loops be detected reliably using timeouts? How can a query scope be used to exploit topology characteristics in answering a query? For improved efficiency, how can queries be executed in containers that concentrate distributed P2P database nodes into hosting environments with virtual nodes?*

- *Can we devise a unified P2P database framework for general-purpose query support in large heterogeneous distributed systems spanning many administrative domains? More precisely, can we devise a framework that is unified in the sense that it allows to express specific applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options?*

In this chapter, we take the first steps towards unifying the fields of database management systems and P2P computing, which so far have received considerable, but separate, attention. We extend database concepts and practice to cover P2P search. Similarly, we extend P2P concepts and practice to support powerful general-purpose query languages such as XQuery [18] and SQL [19]. As a result, we answer the above questions by proposing the so-called *Unified Peer-to-Peer Database Framework (UPDF)*.

The related but orthogonal concepts of (logical) link topology and (physical) node deployment model are introduced. Definitions are proposed, clarifying the notion of node, service, fat, thin and ultra-thin P2P networks, as well as the commonality of a P2P network and a P2P network for service discovery. The agent P2P model is proposed and compared with the servent P2P model. A timeout-based mechanism to reliably detect and prevent query loops is proposed. Four techniques to return matching query results to an originator are characterized, namely *Routed Response*, *Direct Response*, *Routed Metadata Response*, and *Direct Metadata Response*. Query processing in centralized, distributed and P2P databases is unified. A theory of query processing for queries that are (or are not) *recursively partitionable* is proposed, which directly reflects the basis of the P2P scalability potential. The definition and properties of simple, medium and complex queries are clarified with respect to recursive partitioning. It is established to what extent simple, medium and complex queries support pipelining. To ensure that a maximum of results can be delivered reliably within the time frame desired by a user even if a query does not support pipelining, *dynamic abort timeouts* using as policy *exponential decay with halving* are proposed. The result is established that a loop timeout must be static. The concept of query scope is used to navigate and prune the link topology and filter on attributes of the deployment model. Indirect specification of scope based on neighbor selection, timeout and radius is detailed. For improved efficiency, the concept of containers for centralized virtual node hosting is established. Three alternative query execution strategies are proposed, namely *normal query execution*, *collecting traversal* and *quick scope violating query*. The most efficient strategy relaxes the conditions imposed by the query scope, whereas the others preserve the semantics of query and query scope.

## 6.1 Introduction

**Gnutella.** We now repeat Pandurangan's summary of the Gnutella network as introduction [87]: Gnutella is a public P2P network on the Internet, by which anyone can share, search for and retrieve files and content. A participant first downloads one of the available (free) implementations of the search *servent*. The participant may choose to make some documents (say, all his FOCS papers) available for public sharing, and indexes the content of these documents and runs a search server on the index. His servent joins the network by connecting to a small number (typically 3-5) of neighbors currently connected to the network. When any servent `s` wishes to search the network with some query `q`, it sends `q` to its neighbors. These neighbors return any of their own documents that match the query; they also forward `q` to their neighbors, and so on. To control network traffic this fanning-out typically continues to some fixed radius (in Gnutella, typically 7); matching results are fanned back into `s` along the paths on which `q` flowed outwards. Thus, every servent can initiate, forward and serve

query results; clearly it is important that the content being searched for be within the search radius of `s`. A servent typically stays connected for some time, and then drops out of the network – many participating hosts are personal computers on dialup connections.

**Topology.** A servent can be seen as a node in a network. A *link topology* describes the link structure among nodes. It describes which nodes are linked with which other nodes. The simplest topology model is a star. All nodes are connected to a single central node. Several link topology models covering the spectrum from centralized models to fine-grained fully distributed models can be envisaged, among them single node, star, ring, tree, semi hierarchical as well as graph models. Real-world distributed systems often have a more complex organization than any simple topology. They often combine several topologies into a hybrid topology. Nodes typically play multiple roles in such a system. For example, a node might have a centralized interaction with one part of the system, while being part of a hierarchy in another part [41]. Figure 6.1 depicts some example topologies.

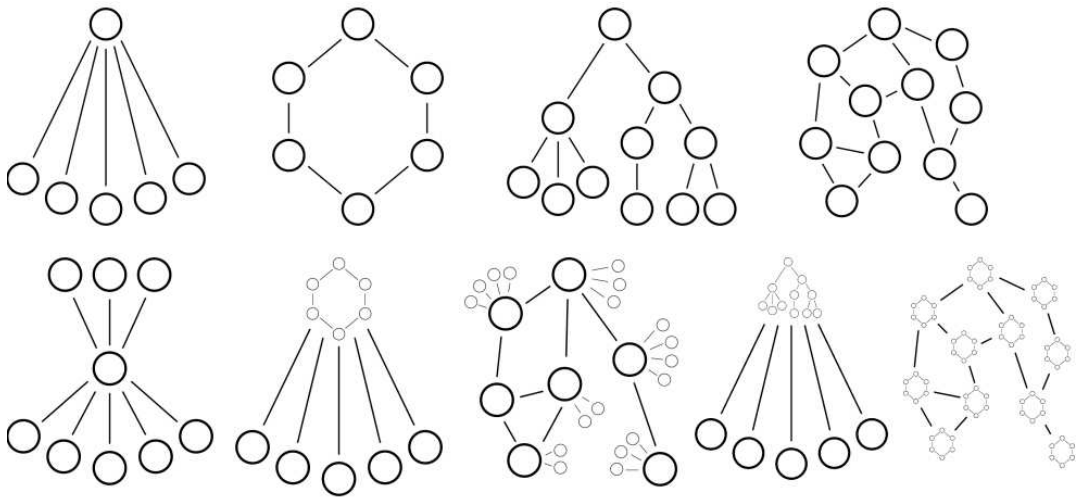

Figure 6.1: Example Link Topologies [41].

Clearly not all nodes in a topology are equal. For example, node bandwidth may vary by four orders of magnitude (50Kbps-1000Mbps), latency by six orders of magnitude (10us-10s), and availability by four orders of magnitude (1%-99.99%).

We stress that it is by no means justifiable to advocate the use of graph topologies irrespective of application context and requirements. Depending on the context, all topologies have their merits and drawbacks in terms of scalability, reliability, availability, content coherence, fault tolerance, security and maintainability. However, from the structural perspective, the graph topology is the most general one, being able to express all other conceivable topologies. Since our goal is to support queries that are generally independent of the underlying topology, this thesis discusses problems arising in graph topologies. A problem solution that applies to a graph also applies to any other topology[1]. The results of this thesis help *enable*

[1]Of course, a simpler or more efficient solution may exist for any particular topology.

the use of graph topologies where appropriate, they do not *require* or *mandate* the use of them.

A link topology is purely a logical construct, as it does *not* describe where and how the link information is stored and accessed. This is defined by a *node deployment model*, which defines where and how one or more partitions of the graph are running, stored and accessed. Consider the analogy to the WWW: The hyperlink topology remains identical, no matter whether all pages of the universe are served by a single large dynamic web server or any kind of worldwide federation of static web servers. A detailed discussion is given in Section 6.9.

**Definitions - Service and Node.** Let us clarify the notion of node and service. A *service* exposes some functionality in the form of service interfaces to remote clients. Example services are an echo service, a job scheduler, a replica catalog, a time service, a gene sequencing service and a language translation service. A *node* is a service that exposes *at least* functionality (i.e. service interfaces) for publication and P2P queries. Examples are a *hyper registry* as introduced in the previous chapter, a Gnutella file sharing node and an extended job scheduler. Put another way, any service that happens to support publication and P2P query interfaces is a node. This implies that *every node is a service*. It does not imply that every service is a node. Only nodes are part of the P2P topology, while services are not, because they do not support the required interfaces. Usually, most services are not nodes. However, in some networks most or all services are nodes.

- **Most services are not nodes.** We propose to speak of a *fat* P2P network. Typically, only one or a few large and powerful services are nodes, enabling publication and P2P queries. An example is a backbone network of 10 large registry nodes that ties together 10 administrative domains, each hosting a registry node to which local domain services can publish to be discovered, as depicted in Figure 6.2 (left). The services (shown small on the edges) are not part of the network. This is the more conventional scenario. Existing legacy services need not be changed at all.

- **Most or all services are nodes.** We propose to speak of a *thin* or *ultra-thin* P2P network. An example is a network of millions of small services, each having some proprietary core functionality (e.g. replica management optimization, gene sequencing, multi-lingual translation), actively using the network for searching (e.g. to discover replica catalogs, remote gene mappers or language dictionary services), but also actively contributing to its search capabilities, as depicted in Figure 6.2 (right). This scenario may appear to be intrusive, as it suggests that legacy services have to be rewritten. However, this is not the case. Thanks to the concept of service descriptions and service interfaces (a service may consist of a set of independent and distributed service interfaces), additional external publication and search functionality can be added to existing legacy services without touching their code base (see Section 2.2).

There is no difference between these scenarios in terms of technology. Discussion in this chapter is applicable to ultra-thin, thin and fat P2P networks. For simplicity of exposition, examples illustrate ultra-thin networks (every service is a node).

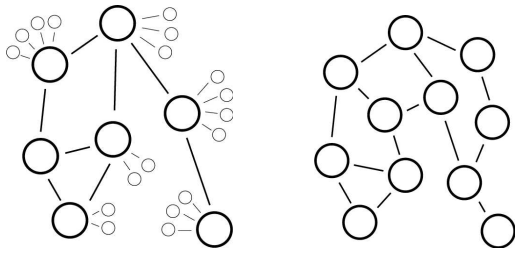

Figure 6.2: Fat (left) and Ultra-thin (right) Peer-to-Peer network.

**Definitions - P2P network vs. P2P network for service discovery.** In any kind of P2P network, nodes may publish themselves to other nodes, thereby forming a topology. In a P2P network for service discovery, services and other content providers may publish their service link and content links to nodes. Because nodes are services, also nodes may publish their service link (and content links) to other nodes, thereby forming a topology. In any kind of P2P network, a node has a database or some kind of data source against which queries are applied. In a P2P network for service discovery, this database happens to be the publication database. In other words, publication enables topology construction and at the same time constructs the database to be searched. Discussion in this chapter is applicable to any kind of P2P network, while the examples illustrate service discovery.

## 6.2 Agent P2P Model and Servent P2P Model

**Agent P2P Model.** Queries in what we propose as the *agent P2P model* flow as follows. When any *originator* wishes to search the P2P network with some query, it sends the query to a single node. We call this entry point the *agent* node of the originator[2]. The agent applies the query to its local database and returns matching results; it also forwards the query to its neighbor nodes. These neighbors return their local query results; they also forward the query to their neighbors, and so on.

For flexibilty, the protocol between originator and agent is left unspecified. The agent P2P model is a hybrid of centralization and decentralization. It allows fully decentralized infrastructures, yet also allows seamless integration of centralized client-server computing into an otherwise decentralized infrastructure. An originator may embed its agent in the same process (decentralized). However, the originator may just as well choose a remote node as agent (centralized), for reasons including central control, reliability, continuous availability, maintainability, security, accounting and firewall restrictions on incoming connections for originator hosts. For example, a simple HTML GUI may be sufficient to originate queries that are sent to an organization's agent node. Note that only nodes are part of the P2P topology, while the originator is not, because it does not possess the functionality of a node. The agent P2P model provides location and distribution transparency to originators. An

[2]We stress that in our context, the term *agent* has nothing to do with intelligence, will-of-its-own, or mobile code. The term is not only well established in classic networking, but is also used in the distributed artificial intelligence community. We use the term in the sense of a *service gateway*.

originator is unaware that (and how) database tuples are partitioned among nodes. It only communicates with an agent black box. ("*People want something that's logically centralized and physically distributed. But they don't want the weird errors that come with distribution*" - Mark Stuart Day, Cisco Systems).

**Servent P2P Model.** In contrast, in the *servent P2P model* (e.g. Gnutella) there exists no agent concept, but only the concept of a servent. Put another way, the agent is always embedded into the originator process, forming a monolithic servent. This model is decentralized, and it does not allow for some degree of centralization. This restriction appears unjustified. More importantly, it seriously limits the applicability of P2P computing. For example, the Gnutella servent model could not cope with the large connectivity spectrum of the user community, ranging from very low to very high bandwidth. As the Gnutella network grew, it became fragmented because nodes with low bandwidth connections could not keep up with traffic. The idea of *requiring* all functionality to exist at the very edge of the network had to be reconsidered. Eventually, the situation was patched by rendering dumb the low bandwidth servents on the (slow) edges of the network. The notion of centralized reflectors (Gnutella) and super-peers (Morpheus) was (re) invented. A reflector is a powerful high bandwidth gateway for many remote originators with low bandwidth dialup connections. It volunteers to take over the functionality and shield traffic that would normally be carried via low bandwidth servents. However, servents still keep data locally. The agent P2P model naturally covers centralized and decentralized hybrids. Here a powerful node may act as agent for many remote originators. In the remainder of this thesis, we follow the agent P2P model and do not use the term *servent* anymore. The terms *originator*, *node* and *agent (node)* are used instead.

## 6.3 Loop Detection

Query shipping is used to *route* queries through the nodes of the topology. A query remains identical during forwards over hops (unless rewritten or split by a query optimizer). The very same query may arrive at a node multiple times, along distinct routes, perhaps in a complex pattern. Loops in query routes must be detected and prevented. Otherwise, unnecessary or endless multiplication of workloads would be caused. Figure 6.2 depicts topologies with the potential for a query to become trapped in infinite loops.

To enable loop detection, an originator attaches a different transaction identifier to each query, which is a universally unique identifier (UUID). The transaction identifier always remains identical during query forwarding over hops. A node maintains a state table of recent transaction identifiers and returns an error whenever a query is received that has already been seen. For example, this approach is used in Gnutella.

In practice, it is sufficient for the UUID to be unique with exceedingly large probability, suggesting the use of a 128 bit integer computed by a cryptographic hash digest function such as MD5 [88] or SHA-1 [89] over message text, originator IP address, current time and a random number.

## 6.4 Routed vs. Direct Response, Metadata Responses

We propose to distinguish four techniques to return matching query results to an originator: *Routed Response*, *Direct Response*, *Routed Metadata Response*, and *Direct Metadata Response*, as depicted in Figure 6.3. Let us examine the main implications with a Gnutella use case. A typical Gnutella query such as *"Like a virgin"* is matched by some hundreds of files, most of them referring to replicas of the very same music file. Not all matching files are identical because there exist multiple related songs (e.g. remixes, live recordings) and multiple versions of a song (e.g. with different sampling rates). A music file has a size of at least several megabytes. Many thousands of concurrent users submit queries to the Gnutella network. A large fraction of users lives on slow and unreliable dialup connections.

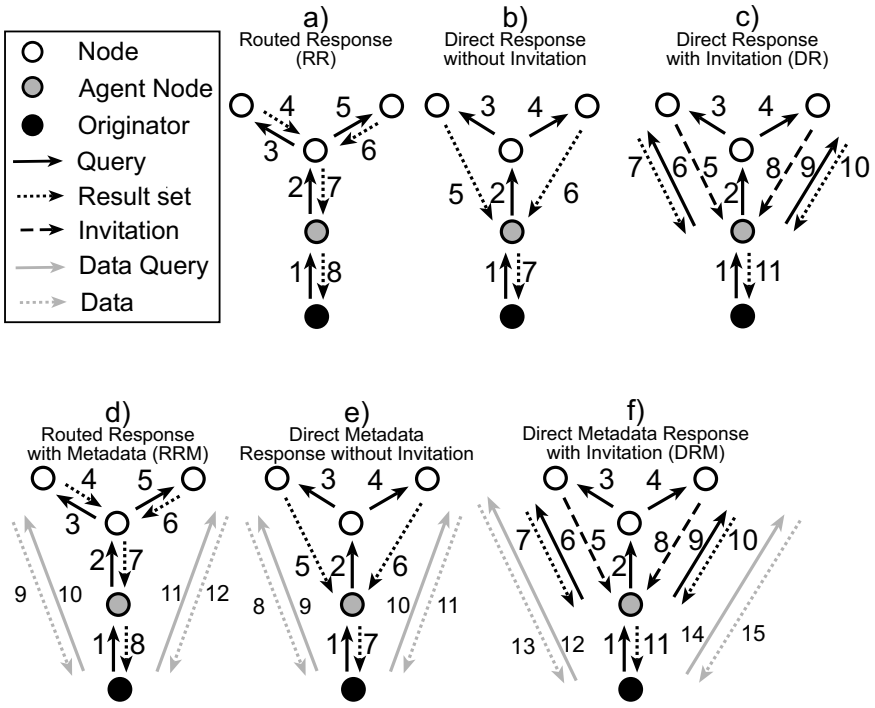


Figure 6.3: Peer-to-Peer Response Modes.

- **Routed Response.** (Figure 6.3-a). Results are propagated back into the originator along the paths on which the query flowed outwards. Each (passive) node returns to its (active) client not only its own local results but also all remote results it receives from neighbors. The response protocol is tightly coupled to the query protocol. Routing messages through a logical overlay network of P2P nodes is much less efficient than routing through a physical network of IP routers [90]. Routing back even a single Gnutella file (let alone all results) for each query through multiple nodes would consume large amounts of overall system bandwidth, most likely grinding Gnutella to a screeching halt. As the P2P network grows, it is fragmented because nodes with low bandwidth

connections cannot keep up with traffic [91]. Consequently, routed responses are not well suited for file sharing systems such as Gnutella. In general, *overall economics* dictate that routed responses are not well suited for systems that return many and/or large results.

- **Direct Response With and Without Invitation.** To better understand the underlying idea, we first introduce the simpler variant, which is Direct Response Without Invitation (Figure 6.3-b). Results are not returned by routing back through intermediary nodes. Each (active) node that has local results sends them directly to the (passive) agent, which combines and hands them back to the originator. Response traffic does not travel through the P2P system. It is offloaded via individual point-to-point data transfers on the edges of the network. The response push protocol can be separated from the query protocol. For example, HTTP, FTP or other protocols may be used for response push. Let us examine the main implications with a use case.

  As already mentioned, a typical Gnutella query such as *"Like a virgin"* is matched by some hundreds of files, most of them referring to replicas of the very same music file. For Gnutella users it would be sufficient to receive just a small subset of matching files. Sending back *all* such files would unnecessarily consume large amounts of direct bandwidth, most likely restricting Gnutella to users with excessive cheap bandwidth at their disposal. Note however, that the overall Gnutella system would be only marginally affected by a single user downloading, say, a million music files, because the largest fraction of traffic does not travel through the P2P system itself.

  In general, *individual economics* dictate that direct responses without invitation are not well suited for systems that return many equal and/or large results, while a small subset would be sufficient. A variant based on invitation (Figure 6.3-c) softens the problem by inverting control flow. Nodes with matching files do not blindly push files to the agent. Instead they invite the agent to initiate downloads. The agent can then act as it sees fit. For example, it can filter and select a subset of data sources and files and reject the rest of the invitations. Due to its inferiority, the variant without invitation is not considered any further. In the remainder of this thesis, we use the term Direct Response as a synonym for Direct Response With Invitation.

- **Routed Metadata Response and Direct Metadata Response.** Here interaction consists of two phases. In the first phase, routed responses (Figure 6.3-d) or direct responses (Figure 6.3-e,f)) are used. However, nodes do not return data results in response to queries, but only small metadata results. The metadata contains just enough information to enable the originator to retrieve the data results and possibly to apply filters before retrieval. In the second phase, the originator selects, based on the metadata, which data results are relevant. The (active) originator directly connects to the relevant (passive) data sources and asks for data results. Again, the largest fraction of response traffic does not travel through the P2P system. It is offloaded via individual point-to-point data transfers on the edges of the network. The retrieval protocol can be separated from the query protocol. For example, HTTP, FTP or other protocols may be used for retrieval.

The routed metadata response approach is used by file sharing systems such as Gnutella. A Gnutella query does not return files; it just returns an annotated set of HTTP URLs. The originator connects to a subset of these URLs to download files as it sees fit. Another example is a service discovery system where the first phase returns a set of service links instead of full service descriptions. In the second phase, the originator connects to a subset of these service links to download service descriptions as it sees fit. Another example is a *referral* system where the first phase uses routed metadata response to return the service links of the set of nodes having local matching results ("*Go ask these nodes for the answer*"). In the second phase, the originator or agent connects directly to a subset of these nodes to query and retrieve result sets as it sees fit. This variant avoids the "invitation storm" possible under Direct Response. Referrals are also known as *redirections*[3].

**Comparison of Response Mode Properties.** Let us compare the properties of the various response models. The following abbreviations are used. RR ... Routed Response, RRM ... Routed Response with metadata, RRX ... Routed Response with and without metadata, DR ... Direct Response, DRX ... Direct Response with and without metadata.

- **Distribution and Location Transparency.** In the response models without metadata, the originator is unaware that (and how) tuples are partitioned among nodes. In other words, these models are transparent with respect to distribution and location. Metadata responses require an originator to contact individual data providers to download full results, and hence are not transparent.

- **(Efficient) Query Support.** All models can answer any query. Both simple and medium queries can be answered efficiently by RRX and DRX, whereas a complex query cannot be answered efficiently. (Justification of this result is deferred to Section 6.5). Transmission of duplicate results unnecessarily wastes bandwidth. RRX can eliminate duplicates already along the query path, whereas DRX can only do so in the final stage, at the agent. Similarly, maximum result set size limiting is more efficient under RRX because superfluous results can already be discarded along the query path.

- **Economics.** RR results travel multiple hops rather than just a single hop. This leads to poor *overall economics*. The effect is more pronounced for large results, as is the case for music files. RR can also lead to unfortunate *individual economics*. A user that induces few or undemanding queries consumes few system resources. However, if many heavy results for queries from other parties are routed back via such a user's node, it can end up in a situation where it pays for large amounts of bandwidth and gives it away for free to anonymous third parties. For a given user, the costs may drastically outweigh the gains. One could perhaps devise appropriate authorization, quality of service and flow control policies. The unsatisfying economic situation is similar to the one of physical IP routers on the Internet, which also forward traffic from and to third

[3]A metadata response mode with a radius scope of zero can be used to implement the referral behavior of the Domain Name System (DNS). For details, see Section 6.11.

parties. In any case, there remains the fact that results travel multiple hops rather than just one.

In principle, RRM has the same poor economic properties as RR. However, if metadata is very small in size (e.g. as in Gnutella), then the incurred processing and transmission cost may be acceptable. For example, Gnutella nodes just route back an annotated set of HTTP URLs as metadata. Under DRX, result traffic does not travel through the P2P system. Retrieving results is a deal between just two parties, the provider and the consumer. Consequently, individual economics are controllable and predictable. A user is not charged much for other peoples workloads, unless he explicitly volunteers.

- **Number of TCP Connections at Originator.** Under RR and DR, just one (or no) TCP connection is required at the originator, whereas metadata modes require a connection per (selected) data provider. The more data sources are selected, the more heavyweight data retrieval becomes. Metadata modes can encounter serious latency limitations due to the very expensive nature of secure (and even insecure) TCP connection setup. Hence, the approach does not scale well. However, for many use cases this may not be a problem because a client always selects only a small number of data providers (e.g. 10).

- **Number of TCP Connections at Agent.** Usually a node has few neighbors (five to hundreds). Under RRX, one TCP connection per neighbor is required at an agent. Under DRX, additionally a connection per data provider is required. Again, the more data providers exist, the more heavyweight data retrieval becomes. DRX can encounter serious latency limitations due to the very expensive nature of secure (and even insecure) TCP connection setup. For example, a query that finds the total number of services in the domain `cern.ch` should use RRX. Under DRX, it may generate responses from every single node in that domain. Consequently, an agent can face an invitation storm resembling a denial of service attack. On the other hand, the potential to exploit parallelism is large. All data providers can be handled independently in parallel.

- **Latency.** If a query is of a type that cannot support pipelining (see Section 6.6), the latency for the first result to arrive at the originator is always poor. For a pipelined query, the latency for the first result to arrive is small under DRX, because a response travels a single hop only. Under RRX, a response travels multiple hops, and latency increases accordingly. However, the cost of TCP connection setup at originator and/or agent can invert the situation. Under RR, the cost of TCP connection setup to nodes is paid only once (at node publication time), because connections can typically be kept alive until node deregistration. This is not the case under the other response modes.

- **Caching.** Caching is a technique that trades content freshness for response time. RRX can potentially support caching of content from other nodes at intermediate nodes because response flow naturally concentrates and integrates results from many nodes. DRX nodes return results directly and independently, and hence cannot efficiently support caching.

- **Trust Delegation to Unknown Parties.** Query and result traffic are subject to security attacks. It is not sufficient to establish a secure mutually authenticated channel between any two nodes because malicious nodes can divert routes or modify queries and results. Since a query is almost always routed through multiple hops, many of which are unknown to the agent, we believe that indirect delegation of trust to unknown parties cannot practically be avoided[4]. Security sensitive applications should choose DRX because at least the retrieval of results occurs in a predictable manner between just two parties that can engage in secure mutual authentication and authorization. RRM merely delegates trust on metadata results, but not on full results. The properties discussed are summarized in Table 6.1.

| | **Routed Response** | **Direct Response** | **Any Metadata Response** |
|---|---|---|---|
| *Query supported* | Any | Any | Any |
| *Query efficiently supported* | Simple, Medium | Simple, Medium | n.a. |
| *Duplicate elimination* | Early | Late | n.a. |
| *Maximum result set size limiting* | Early | Late | n.a. |
| *Messages traveling in P2P system / Risk to pay more than earn* | Query and large results / Large | Query / Small | Metadata instead of results / small |
| *First result latency assuming pipelining* | N hops, no connection setup | 1 hop, connection setup | n.a. |
| *TCP connections at originator / Scalability in the number of nodes having results* | One / Good | One / Good | Large / Poor |
| *TCP connections at agent / parallelism / Scalability in the number of nodes having results* | Small / Small / Good | Large / Large / Poor | n.a. |
| *Caching possible* | Yes | No | n.a. |
| *Trust delegation to unknown parties* | Query and Results | Query | + Metadata |

Table 6.1: Comparison of Peer-to-Peer Response Mode Properties

**Response Mode Switches and Shifts.** Although from the functional perspective all response modes are equivalent, clearly no mode is optimal under all circumstances. The question arises as to what extent a given P2P network must mandate the use of any particular response mode throughout the system. Observe that nodes are autonomous and defined by their interface only. A node does not "see" what kind of response mode (or technology in general) its neighbors use in answering a query. As long as query semantics are preserved, the node does not care. Consequently, we propose that response modes can be mixed by *switches* and *shifts*, in arbitrary permutations, as depicted in Figure 6.4.

- **Routed Response ⇒ Direct Response switch.** (Figure 6.4-a). Starting from the agent, Routed Response is used initially. The central node ("football") receives a query

[4]Even though trust delegation technologies exist [92], they do not scale to a significant number of autonomous parties, let alone parties that dynamically join and leave. The problem is how to enable practical establishment and administration of direct and indirect trust relationships.

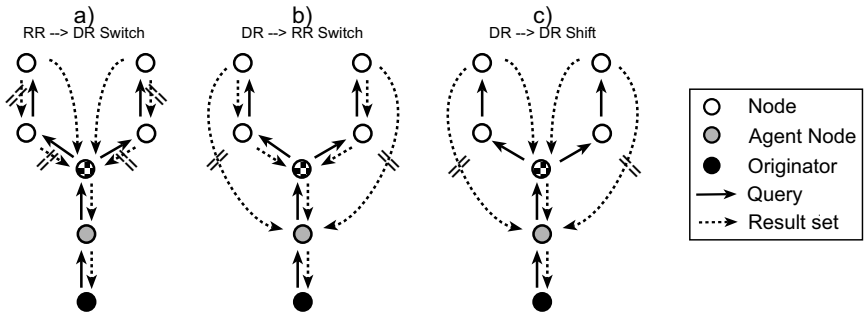


Figure 6.4: Response Mode Switches and Shifts.

from the agent. For some reason, it decides to answer the query using Direct Response. The response flow that would have been taken under Routed Response is shown crossed out.

- **Direct Response ⇒ Routed Response switch.** (Figure 6.4-b). Initially, Direct Response is used. However, the "football" decides to answer the query using Routed Response.

- **Direct Response ⇒ Direct Response shift.** (Figure 6.4-c). Initially, Direct Response is used. The football decides to continue using Direct Response but shift the target of responses. To its own neighbors the football declares itself as (a fake) agent. The responses that would have flowed into the real agent now flow back into the football, and then from the football to the real agent. Note again that this does not break semantics because the football behaves as if the results would have been obtained from its own local database. The real agent receives the same results, but solely from the football.

- **Routed Response ⇒ Routed Response shift.** At each hop, the response target is shifted to be the current node. Interestingly, this kind of shift is at the very heart of the definition of routed response. The classification introduced here shows that this is not the only possible approach.

A node may choose its response mode based on a local and autonomous assessment of the advantages and disadvantages involved. However, because of its context knowledge, often the client (e.g. originator) is in the best position to judge what kind of response mode would be most suitable. Therefore, it is useful to allow specifying as part of the query a hint that indicates the preferred response mode (`routed` or `direct`).

## 6.5 Query Processing

In a distributed database system, there exists a single local database and zero or more neighbors. A classic centralized database system is a special case where there exists a single local

database and zero neighbors. From the perspective of query processing, a P2P database system has the same properties as a distributed database system, in a recursive structure.

Hence, we propose to organize the P2P query engine like a general distributed query engine [45, 93]. A given query involves a number of operators (e.g. SELECT, UNION, CONCAT, SORT, JOIN, GROUP, SEND, RECEIVE, SUM, MAX, MAXSETSIZE, IDENTITY) that may or may not be exposed at the query language level. For example, the SELECT operator takes a set and returns a new set with tuples satisfying a given predicate. The UNION operator computes the union of two or more sets. The CONCAT operator concatenates the elements of two or more sets into a list of arbitrary order (without eliminating duplicates)[5]. The MAXSETSIZE operator limits the maximum result set size. The IDENTITY operator returns its input set unchanged.

The semantics of an operator can be satisfied by several operator implementations, using a variety of algorithms, each with distinct resource consumption, latency and performance characteristics. The query optimizer chooses an efficient query execution plan, which is a tree plugged together from operators. In an execution plan, a parent operator consumes results from child operators. Query execution is driven from the (final) root consumer in a top down fashion. For example, a request for results from the root operator may in turn lead to a request for results from child operators, which in turn request results from their own child operators, and so on. By means of an execution plan, an optimizer can move a query to data or data to a query. In other words, queries and sub queries can be executed locally or at remote nodes. Performance tradeoffs of query shipping, data shipping and hybrid shipping are discussed in [94].

**Template Query Execution Plan.** Recall that Section 3.2 proposed a query model. *Any* query `Q` within our query model can be answered by an agent with the *template execution plan* `A` depicted in Figure 6.5. The plan applies a local query `L` against the tuple set of the local database. Each neighbor (if any) is asked to return a result set for (the same) neighbor query `N`. Local and neighbor result sets are unionized into a single result set by a unionizer operator `U` that must take the form of either `UNION` or `CONCAT`. A merge query `M` is applied that takes as input the result set and returns a new result set. The final result set is sent to the client, i.e. another node or an originator.

**Centralized Execution Plan.** To see that indeed any query against any kind of database system can be answered within this framework we derive a simple *centralized execution plan* that always satisfies the semantics of any query `Q`. The plan substitutes specific subplans into the template plan `A`, leading to distinct plans for the agent node (Figure 6.6-a) and neighbors nodes (Figure 6.6-b). More specifically, in the case of XQuery and SQL, parameters are substituted as follows:

[5] A list can be emulated by a set using distinct surrogate keys.

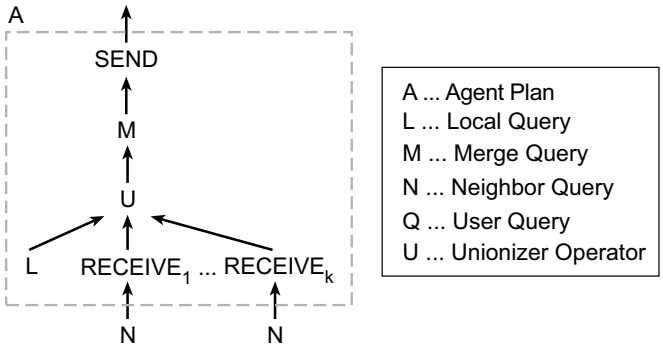


Figure 6.5: Template Execution Plan.

| XQuery | SQL |
|---|---|
| A: M = Q<br>U = UNION<br>L = "RETURN /"<br>N' = N<br><br>N: M = IDENTITY<br>U = UNION<br>L = "RETURN /"<br>N' = N | A: M = Q<br>U = UNION<br>L = "SELECT *"<br>N' = N<br><br>N: M = IDENTITY<br>U = UNION<br>L = "SELECT *"<br>N' = N |

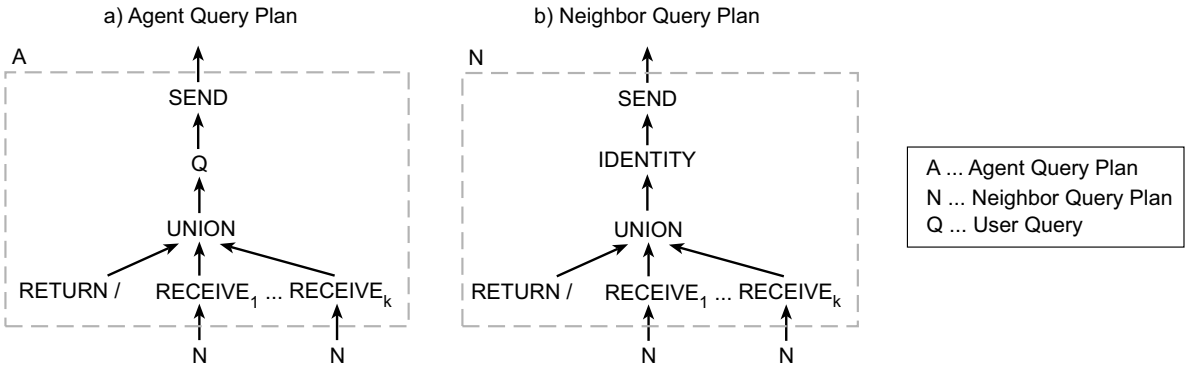


Figure 6.6: Centralized Execution Plan. The Agent Query Plan (a) fetches all raw tuples from the local and all remote databases, unionizes the result sets, and then applies the query `Q`. Neighbors are handed a rewritten neighbor query (b) that recursively fetches all raw tuples, and returns their union.

In other words, the agent's plan `A` fetches all raw tuples from the local and all remote databases, unionizes the result sets, and then applies the query `Q`. Neighbors are handed a rewritten neighbor query `N` that recursively fetches all raw tuples, and returns their union. The neighbor query `N` is recursively partitionable (see below).

The same centralized plan works for routed and direct response, both with and without metadata. Under direct response, a node does forward the query `N`, but does not attempt to receive remote result sets (conceptually empty result sets are delivered). The node does not send a result set to its predecessor, but directly back to the agent.

In a distributed database system, there exists a single local database and zero or more neighbors. A classic centralized database system is a special case where there exists a single local database and zero neighbors. From the perspective of query processing, a P2P database system has the same properties as a distributed database system, in a recursive structure. Consequently, the very same centralized execution plan applies to any kind of database system; and any query within our query model can be answered.

The centralized execution plan can be inefficient because potentially large amounts of base data have to be shipped to the agent before locally applying the user's query. However, sometimes this is the only plan that satisfies the semantics of a query. This is always the case for a complex query. A more efficient execution plan can sometimes be derived (as proposed below). This is always the case for a simple and medium query.

**Recursively Partitionable Query.** A P2P network can be efficient in answering queries that are recursively partitionable. A query `Q` is *recursively partitionable* if, for the template plan `A`, there exists a merge query `M` and a unionizer operator `U` to satisfy the semantics of the query `Q` assuming that `L` and `N` are chosen as `L = Q` and `N = A`. In other words, a query is recursively partitionable if the very same execution plan *can* be recursively applied at every node in the P2P topology. The corresponding execution plan is depicted in Figure 6.7.

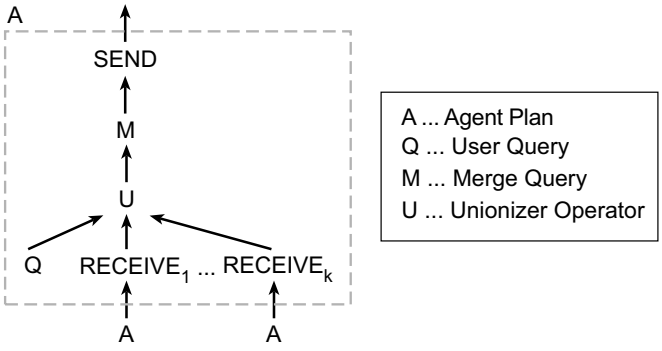


Figure 6.7: Execution Plan for Recursively Partitionable Query.

The input and output of a merge query have the same form as the output of the local query `L`. Query processing can be parallelized and spread over all participating nodes. Potentially very large amounts of information can be searched while investing little resources such as processing time per individual node. The recursive parallel spread of load implied by a recursively partitionable query is the basis of the massive P2P scalability potential. However, query performance is not necessarily good, for example due to high network I/O costs.

Now we are in the position to clarify the definition of simple, medium and complex queries.

- *Simple Query.* A query is *simple* if it is recursively partitionable using `M = IDENTITY`,

`U = UNION` (e.g. *QS1 - QS7, Q20 - Q22*).

- *Medium Query.* A query is a *medium* query if it is not simple, but it is recursively partitionable (*QM1 - QM5* and *Q25 - Q26*).

- *Complex Query.* A query is *complex* if it is not recursively partitionable (e.g. *QC1 - QC3*).

It is an interesting open question (at least to us) if a query processor can automatically determine whether a correct merge query and unionizer exist, and if so, how to choose them. Related problems have been studied extensively in the context of distributed and parallel query processing as well as query rewriting for heterogeneous and homogenous relational database systems [45, 95, 93]. Distributed XQueries are an emerging field [96]. For simplicity, in the remainder of this thesis we assume that the user explicitly provides `M` and `U` along with a query `Q`. If `M` and `U` are not provided as part of a query to any given node, the node acts defensively by assuming that the query is not recursively partitionable. Choosing `M` and `U` is straightforward for a human being. Consider for example the following medium XQueries.

- *(QM2) Return the number of replica catalogs services.* The merge query computes the sum of a set of numbers[6]. The unionizer is CONCAT.

```
Q = RETURN
      <tuple>
        count(/tupleset/tuple/content/service[interface/@type="repcat"])
      </tuple>
M = RETURN
      <tuple>
        sum(/tupleset/tuple)
      </tuple>
U = CONCAT
```

- *(QM1) Find the service with the largest uptime.*

```
Q=M= RETURN (/tupleset/tuple[@type="service"] SORTBY (./@uptime)) [last()]
U =  UNION
```

A custom merge query can be useful. For example, assume that each individual result tuple is tagged with a timestamp indicating the time when the information expires and ceases to be valid. A custom merge query can ignore all results that have already expired. Alternatively, it can ignore all results but the one with the most recent timestamp. As another example, a custom merge query can cut off all but the first 100 result tuples. Such a result set size limiting feature (`maxResults`) attempts to make bandwidth consumption more predictable.

[6]Recall from Section 3.3 that the query engine always encapsulates the query output with a `tupleset` root element. A query need not generate this root element as it is implicitly added by the environment.

## 6.6 Pipelining

The success of many applications depends on how fast they can start producing initial/relevant portions of the result set rather than how fast the entire result set is produced [97]. This is particularly often the case in distributed systems where many nodes are involved in query processing, each of which may be unresponsive for many reasons. The situation is even more pronounced in systems with loosely coupled autonomous nodes.

Often an originator would be happy to already do useful work with one or a few *early results*, as long as they arrive quickly and reliably. Results that arrive later can be handled later, or are ignored anyway. For example, in an interactive session, a typical Gnutella user is primarily interested in being able to start *some* music download as soon as possible. The user is quickly disappointed when not a single result for a query arrives in less than four seconds. Choosing among 1000 species of "Like a virgin" is interesting, but helps little if it comes at the expense of, say, one minute idle waits. As another example, consider a user that wants to discover schedulers to submit a job. It is interesting to discover that 100 schedulers are available, but the primary requirement is to find at least three quickly and reliably.

Database theory and practice establishes that query execution engines in general, and distributed query execution engines in particular should be based on iterators [45]. An operator corresponds to an iterator class. Iterators of any kind have a uniform interface, namely the three methods `open()`, `next()` and `close()`. In an execution plan, a parent iterator consumes results from child iterators. Query execution is driven from the (final) root consumer in a top down fashion. For example, a call to `next()` may call `next()` on child iterators, which in turn call `next()` on their child iterators, and so on. For efficiency, the method `next()` can be asked to deliver several results at once in a so-called *batch*. Semantics are as follows: *"Give me a batch of at least* `N` *and at most* `M` *results"* (less than `N` results are delivered when the entire query result set is exhausted). For example, the SEND and RECEIVE network communication operators (iterators) typically work in batches.

The monotonic semantics of certain operators such as SELECT, UNION, CONCAT, SEND, RECEIVE, MAXSETSIZE, IDENTITY allow that operator implementations consume just one or a few child results on `next()`. In contrast, the non-monotonic semantics of operators such as SORT, GROUP, MAX, some JOIN methods, etc. require that operator implementations consume *all* child results already on `open()` in order to be able to deliver a result on the first call to `next()`. Since the output of these operators on a subset of the input is not, in general, a subset of the output on the whole input, these operators need to see all of their input before they produce the correct output. This does not break the iterator concept but has important latency and performance implications. Whether the root operator of an agent exhibits a short or long latency to deliver to the originator the first result from the result set depends on the query operators in use, which in turn depend on the given query. In other words, for some query types the originator has the potential to immediately start piping in results (at moderate performance rate), while for other query types it must wait for a long time until the first result becomes available (the full result set arrives almost at once, however).

A query (an operator implementation) is said to be *pipelined* if it can already produce at least one result tuple before all input tuples have been seen. Otherwise, a query (an operator)

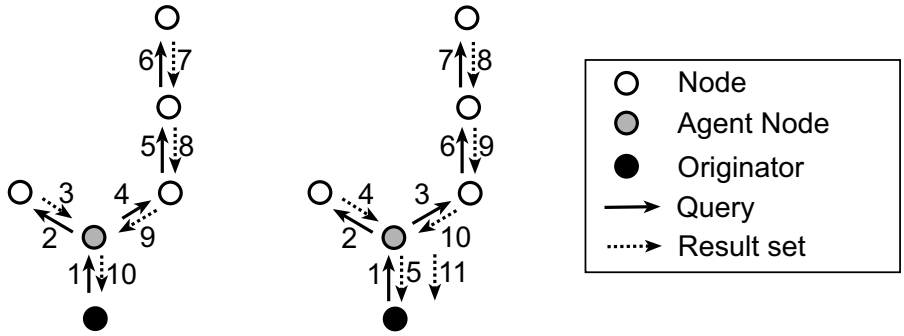


Figure 6.8: Non-Pipelined (left) and Pipelined Query (right).

| Query Type | Supports Pipelining? |
|---|---|
| Simple Query | Yes |
| Medium Query | Maybe |
| Complex Query | Typically No |

Table 6.2: Pipelining Support of Query Types.

is said to be *non-pipelined*. Figure 6.8 depicts examples for both modes.

Simple queries do support pipelining (e.g. Gnutella queries). Medium queries may or may not support pipelining, whereas complex queries typically do not support pipelining. The properties are summarized in Table 6.2.

Bear in mind, that even if a query can be pipelined, the messaging model and underlying network layers in use may not support pipelining, in which case a result set has to be delivered with long latency in a single large batch. To this end, Chapter 7 proposes a Peer Database Protocol (as opposed to query operator implementations) that efficiently supports pipelining and non-pipelining, and is applicable to any centralized, distributed or P2P architecture, including routed response and direct response, both with and without metadata modes. Finally note that non-pipelining delivery without a dynamic abort timeout feature is highly unreliable due to the so-called simultaneous abort problem (see below). If only one of the many nodes in the query path fails to be responsive for whatever reasons, all other nodes in the chain are waiting, eventually time out at the same time, and the originator receives not even a single result.

## 6.7 Static Loop Timeout and Dynamic Abort Timeout

Clearly there comes a time when a user is no longer interested in query results, no matter whether any more results might be available. The query roaming the network and its response traffic should fade away after some time. In addition, P2P systems are well advised to attempt to limit resource consumption by defending against *runaway* queries roaming forever or producing gigantic result sets, either unintended or malicious. To address these problems, an absolute *abort timeout* is attached to a query, as it travels across hops. An abort timeout

can be seen as a deadline. Together with the query, a node tells a neighbor *"I will ignore (the rest of) your result set if I have not received it before 12:00:00 today."* The problem, then, is to ensure that a maximum of results can be delivered reliably within the time frame desired by a user.

The value of a *static timeout* remains unchanged across hops, except for defensive modification in flight triggered by runaway query detection (e.g. infinite timeout). In contrast, it is intended that the value of a *dynamic timeout* be decreased at each hop. Nodes further away from the originator may time out earlier than nodes closer to the originator.

**Dynamic Abort Timeout.** A static abort timeout is entirely unsuitable for non-pipelined result set delivery, because it leads to a serious reliability problem, which we propose to call *simultaneous abort timeout.* If just one of the many nodes in the query path fails to be responsive for whatever reasons, all other nodes in the path are waiting, eventually time out and attempt to return at least a partial result set. However, it is impossible that any of these partial results ever reach the originator, because all nodes time out *simultaneously* (and it takes some time for results to flow back). For example, the agent times out and attempts to return its local partial results to the originator. After that, all partial results flowing to the agent from neighbors, and their neighbors, etc. are discarded – it is already too late. However, even the agent cannot deliver results to the originator because the originator has already timed out (shortly) before the results arrive. Hence, the originator receives not even a single result if just one of the many nodes in the query path fails to be responsive.

To address the simultaneous abort timeout problem, we propose dynamic abort timeouts. Under *dynamic abort timeout*, nodes do not time out at the same time. Instead, nodes further away from the originator time out earlier than nodes closer to the originator. This provides some safety time window for the partial results of any node to flow back across multiple hops to the originator. Together with the query, a node tells a neighbor *"I will ignore (the rest of) your result set if I have not received it before 12:00:00 today. Do whatever you think is appropriate to meet this deadline"*. Intermediate nodes can and should adaptively decrease the timeout value as necessary, in order to leave a large enough time window for receiving and returning partial results subsequent to timeout.

Observe that the closer a node is to the originator, the more important it is (because if it cannot meet its deadline, results from a large branch are discarded). Further, the closer

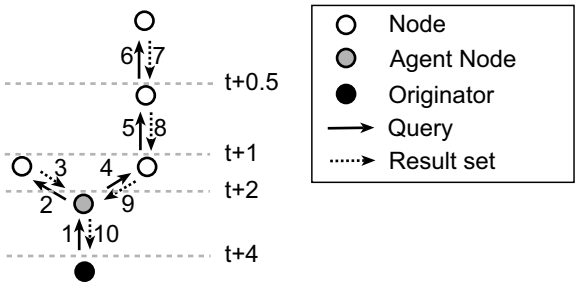


Figure 6.9: Dynamic Abort Timeout.

a node is to the originator, the larger is its response and bandwidth consumption. Thus, as a good policy to choose the safety time window, we propose *exponential decay with halving*. The window size is halved at each hop, leaving large safety windows for important nodes and tiny window sizes for nodes that contribute only marginal result sets. Also, taking into account network latency and the time it takes for a query to be locally processed, the timeout is updated at each hop `N` according to the following recurrence formula:

$$timeout_N = currenttime_N + \frac{timeout_{N-1} - currenttime_N}{2} \tag{6.1}$$

Consider for example Figure 6.9. At time `t` the originator submits a query with a dynamic abort timeout of `t+4` seconds. In other words, it warns the agent to ignore results after time `t+4`. The agent in turn intends to safely meet the deadline and so figures that it needs to retain a safety window of 2 seconds, already starting to return its (partial) results at time `t+2`. The agent warns its own neighbors to ignore results after time `t+2`. The neighbors also intend to safely meet the deadline. From the 2 seconds available, they choose to allocate 1 second, and leave the rest to the branch remaining above. Eventually, the safety window becomes so small that a node can no longer meet a deadline on timeout. The results from the unlucky node are ignored, and its partial results are discarded. However, other nodes below and in other branches are unaffected. Their results survive and have enough time to hop all the way back to the originator before time `t+4`.

Instead of ignoring results which miss their deadline a node may also close the connection. This may, but need not, be harmless. The connection is typically simply reestablished as soon as a new query is to be forwarded. However, in an attempt to educate good P2P citizens, a node may choose to stop propagating or deny service to neighbors that repeatedly do not meet abort deadlines. For example, a strategy may use an exponential back-off algorithm. Note that as long as a node obeys its timeout it can independently implement any timeout policy it sees fit for its purposes without regard to the policy implemented at other nodes. If a node misbehaves or maliciously increases the abort timeout, it risks not being able to meet its own deadline, and is likely soon dropped or denied service. Such healthy measures move less useful nodes to the edge of the network where they cause less harm, because their number of topology links tends to decrease.

To summarize, under non-pipelined result set delivery, dynamic abort timeouts using *exponential decay with halving* ensure that a maximum of results can be delivered reliably within the time frame desired by a user. We speculate that dynamic timeouts could also incorporate sophisticated cost functions involving latency and bandwidth estimation and/or economic models.

**Static Loop Timeout.** Interestingly, a static loop timeout is required in order to fully preserve query semantics. A dynamic timeout (e.g. the dynamic abort timeout) is unsuitable to be used as loop timeout. Otherwise, a problem arises that we propose to call *non-simultaneous loop timeout.* Recall from Section 6.3 that the same query may arrive at a node multiple times, along distinct routes, perhaps in a complex pattern. Loops in query routes must be detected and prevented. Otherwise, unnecessary or endless multiplication of workloads would be caused. To this end, a node maintains a state table of recent transaction

identifiers and associated *loop timeouts* and returns an error whenever a query is received that has already been seen (according to the state table). Before the loop timeout is reached, the same query can potentially arrive multiple times, along distinct routes. On loop timeout, a node may "forget" about a query by deleting it from the state table. To be able to reliably detect a loop, a node must not forget a transaction identifier before its loop timeout has been reached.

However, let us assume for the moment that a dynamic timeout (e.g. the dynamic abort timeout) is used as loop timeout. Consider for example, Figure 6.10, which is identical to Figure 6.9 except that the agent has an additional neighbor that can potentially receive the query along more than one path. At time `t` the originator submits a query with a dynamic abort timeout of `t+4` seconds. The agent in turn warns its own neighbors to ignore results after time `t+2`. Request 5 is sent and arrives, is processed, and its results (step 8) are delivered before the dynamic abort timeout of time `t+1`. At time `t+1` the loop timeout is reached and the query is deleted from the state table. For many reasons, including temporary network segment problems and sequential neighbor processing, request 10 can be delayed. In the example it arrives after time `t+1`. By this time, the receiving node has already forgotten that it already handled the very same query. Hence, the node cannot detect the loop and continues to process and forward (step 11, 12) the same query again.

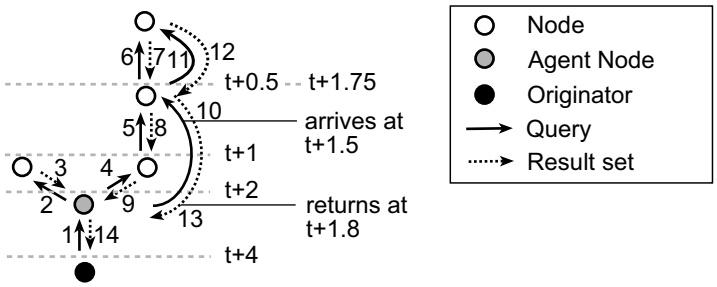


Figure 6.10: Loop Detection Failure with Dynamic Loop Timeout.

The non-simultaneous loop timeout problem is caused by the fact that some nodes still forward the query to other nodes when the destinations have already forgotten it. In other words, the problem is that loop timeout does not occur simultaneously everywhere. Consequently, a loop timeout must be static (does not change across hops) to guarantee that loops can reliably be detected. Along with a query, an originator not only provides a dynamic abort timeout, but also a static loop timeout. Initially at the originator, both values must be identical (e.g. `t+4`). After the first hop, both values become unrelated.

To summarize, we have `abort timeout` $\leq$ `loop timeout.` Loop timeouts must be static whereas abort timeouts may be static or dynamic. Under non-pipelined result set delivery, dynamic abort timeouts using *exponential decay with halving* ensure that a maximum of results can be delivered reliably within the time frame desired by a user. A dynamic abort timeout model still requires static loop timeouts to ensure reliable loop detection, so that a node does not forward and answer the same query multiple times.

Finally, note that to allow for meaningful comparison, sources generating time stamps and sinks processing time stamps must be synchronized, for example using the NTP network time protocol [72]. Further, nodes in the system must share a common representation of time. We subsequently assume a straightforward standard representation: the difference, measured in milliseconds, between the given UTC time and midnight, January 1, 1970 UTC.

## 6.8 Query Scope

As in a data integration system, the goal is to exploit several independent information sources as if they were a single source. This is important for distributed systems in which node topology or deployment model change frequently. For example, cross-organizational Grids and P2P networks exhibit such a character. However, in practice, it is often sufficient (and much more efficient) for a query to consider only a subset of all tuples (service descriptions) from a subset of nodes. For example, a typical query may only want to search tuples (services) within the scope of the domain `cern.ch` and ignore the rest of the world.

Recall that to this end, Section 3.2 cleanly separated the concepts of (logical) query and (physical) query scope. A query is formulated against a global database view and is insensitive to link topology and deployment model. In other words, to a query the set of tuples appears as a single homogenous database, even though the set may be (recursively) partitioned across many nodes and databases. This means that in a relational or XML environment, at the global level, the set of all tuples appears as a single, very large, table or XML document, respectively. The query scope, on the other hand, is used to navigate and prune the link topology and filter on attributes of the deployment model. Conceptually, the scope is the input fed to the query. The query scope is a set and may contain anything from all tuples in the universe to none. Both query and scope can prune the search space, but they do so in a very different manner.

A query scope is specified either *directly* or *indirectly*. For example, one can directly enumerate the tuples (service descriptions) to be considered. However, this is usually impractical. One can also indirectly define a query scope by specifying a set of nodes, implying that the query should be evaluated against the union of all tuples contained in their respective databases. One can distinguish scopes based on neighbor selection, timeout and radius. Note that for security reasons a node may choose to ignore or override a third party provided query scope, for example to guard against runaway queries with infinite scope.

**Neighbor Selection.** For simplicity, all our discussions so far have implicitly assumed a *broadcast* model (on top of TCP) in which a node forwards a query to all neighbor nodes. However, in general one can select a subset of neighbors, and forward concurrently or sequentially. Fewer query forwards lead to less overall resource consumption. The issue is critical because of the snowballing (epidemic, flooding) effect implied by broadcasting. Overall bandwidth consumption grows exponentially with the query radius, producing enormous stress on the network and drastically limiting its scalability. For details, consult [98, 90].

Clearly selecting a neighbor subset can lead to incomplete coverage, missing important results. The best policy to adopt depends on the context of the query and the topology.

Context is *required* to improve on the broadcast model. For example, it makes little sense to forward a Gnutella query to non-Gnutella nodes. The scope can select only neighbors with a service description of interface type "Gnutella". In an attempt to explicitly exploit topology characteristics, a virtual organization of a Grid may deliberately organize global, intermediate and local job schedulers into a tree-like topology. Correct operation of scheduling may require reliable discovery of all or at least most relevant schedulers in the tree. In such a scenario, random selection of half of the neighbors at each node is certainly undesirable. A policy that selects all `child` nodes and ignores all `parent` nodes may be more adequate.

Further, a node may maintain statistics about its neighbors. One may only select neighbors that meet minimum requirements in terms of latency, bandwidth or historic query outcomes (`maxLatency, minBandwidth, minHistoricResult`). Other node properties such as hostname, domain name, owner, etc. can be exploited in query scope guidance, for example to implement security policies. Consider an example where the scheduling system may only trust nodes from a select number of security domains. Here a query should never be forwarded to nodes not matching the trust pattern.

Further, in some systems, finding a single result is sufficient. In general, a user or any given node can guard against unnecessarily large result sets, message sizes and resource consumption by specifying the maximum number of result tuples (`maxResults`) and bytes (`maxResultsBytes`) to be returned. Using sequential propagation, depending on the number of results already obtained from the local database and a subset of the selected neighbors, the query may no longer need to be forwarded to the rest of the selected neighbors.

**Neighbor Selection Query.** For flexibility and expressiveness, we propose to allow the user to specify the selection policy. In addition to the normal query, the user defines a *neighbor selection query* (XQuery) that takes the tuple set of the current node as input and returns a subset that indicates the nodes selected for forwarding. For example, a neighbor query implementing broadcasting selects all services with registry publication and P2P query capabilities, as follows:

```
RETURN /tupleset/tuple[@type="service"
  AND content/service/interface[@type="Consumer-1.0"]
  AND content/service/interface[@type="XQuery-1.0"]]
```

A wide range of policies can be implemented in this manner. The neighbor selection policy can draw from the rich set of information contained in the tuples published to the node. Tuple metadata such as type, context, timestamps, etc. can be used for neighbor selection decisions (see Section 4.3). Further, recall that the set of tuples in a database may not only contain service descriptions of neighbor nodes (e.g. in WSDL or SWSDL), but also other kind of content published from any kind of content provider. For example, this may include host and network information as well as statistics a node periodically publishes to its immediate neighbors.

For example, broadcast and random selection can be expressed with a neighbor query. One can select nodes that support given interfaces (e.g. Gnutella, Freenet or job scheduling). In a tree topology, a policy can use the tuple `context` attribute to select all `child` nodes and to ignore all `parent` nodes. One can implement domain filters and security filters (e.g.

`allow/deny` regular expressions as used in the Apache HTTP server) if the tuple set includes metadata such as hostname and node owner. Power-law policies [29] can be expressed if metadata includes the number of neighbors to the `n`-th radius.

As usual, for security reasons, a node may choose to ignore, override or extend any scope hints it receives. The neighbor query concept can also be used for flexible policy implementation internal to a node. In this case, a node always ignores the user provided neighbor query and uses an internal custom neighbor selection query instead.

**Timeout.** Clearly there comes a time when a user is no longer interested in query results, no matter whether any more results might be available. The query roaming the network and its response traffic should fade away after some time. Section 6.7 already discussed in depth this issue and its implications on loop detection and non-pipelined result set delivery. Here we just note that timeouts clearly belong to the query scope, rather than the query itself.

**Radius.** The *radius* of a query is a measure of path length. More precisely, it is the maximum number of hops a query is allowed to travel on any given path. The radius is decreased by one at each hop. The roaming query and response traffic must fade away upon reaching a radius of less than zero. A scope based on radius serves similar purposes as a timeout. Nevertheless, timeout and radius are complementary scope features. The radius can be used to indirectly limit result set size. In addition, it helps to limit latency and bandwidth consumption and to guard against runaway queries with infinite lifetime. In Gnutella and Freenet, the radius is the primary means to specify a query scope[7]. Neither of these systems support timeouts.

For maximum result set size limiting, a timeout and/or radius can be used in conjunction with neighbor selection, routed response, and perhaps sequential forward, to implement the *expanding ring* [99] strategy. The term stems from IP multicasting. Here an agent first forwards the query to a small radius/timeout. Unless enough results are found, the agent forwards the query again with increasingly large radius/timeout values to reach further into the network, at the expense of increasingly large overall resource consumption. On each expansion radius/timeout are multiplied by some factor.

We now turn to some more subtle points. When precisely does the radius trigger the end of query life? A node rejects a query with a radius less than zero. When a node accepts a query, the radius is decreased by one, and the query is evaluated. The query is not forwarded to neighbors if the new radius is less than zero. In other words, a new radius less than zero forces a neighbor selection policy that yields an empty set. For example, an originator can determine the neighbors of an agent by sending it a query with a radius of zero hops.

Note that the radius is not defined to be *the number* of hops a query is allowed to travel on any given path. Rather, it is more weakly defined to be the maximum number of hops a query is allowed to travel on any given path. In other words, it is not guaranteed that a query takes the shortest path from the agent to any given node, thereby covering a total maximum of nodes. There are two reasons for this kind of definition. First, a node may choose to decrease the radius by any value it sees fit in order to reduce resource consumption or to

[7]The radius is termed *TTL (time-to-live)* in these systems.

prevent system exploitation. Second, loop detection and unpredictable timing in distributed systems can lead to a phenomenon we propose to call *greedy radius pruning*. Recall that the very same query may arrive at a node multiple times, along distinct routes, perhaps in a complex pattern. Traveling `N` hops decreases the radius of a query by `N`. If the query first arrives via a route with many (fast) hops, and later arrives again via a route with few (slow hops), the second arrival will be detected as a loop and rejected. However, the successfully forwarded (first) query continues to travel less hops than theoretically possible considering the (larger) radius of the second query. If the second query had arrived first, the query would have been able to travel further and potentially collect more matching results. Propagating a query to all neighbors concurrently may somewhat increase query coverage, in particular in homogenous LANs.

We conclude this section with a list of more speculative means to indirectly specify a query scope.

**Multiple Entry Nodes.** Suppose an originator using a dumb front end to a central remote agent (e.g. the user of a HTML GUI) knows a set of useful nodes (besides its own agent) as entry points for answering a query. For example, it may know one node in each of the participating laboratories of a virtual organization with global spread. Along with the query, the originator may want to provide to its agent a list of entry nodes for query forward. This kind of behavior seems to be at odds with the spirit of P2P computing because communicating nodes are no longer connected as neighbors. The behavior most likely cannot be emulated by temporarily including the entry nodes as neighbors of the agent, because the originator most likely has no administrative control over the central agent, in particular considering the potential for denial of service attacks. In addition, side effects may occur if multiple originators sharing the same agent modify the set of neighbors. Should the behavior reside in the originator rather than the agent? This appears to be the cleanest approach. If the originator is too limited to implement complex logic, a mediator between originator and agent should be introduced that handles query forward to multiple agents, and subsequent result set merging.

**Path Selection.** One can envision a powerful way to navigate and prune the topology via a path expression, as seen from the originator. This can be of significant practical value. For example, for security reasons one may want to forward a query solely along `cern.ch` node paths, ignoring the rest of the world. In other words, any given node should receive the query only if there exists a continuous path of `cern.ch` nodes from the node back to the agent. This includes nodes from other domains that are directly reachable from `cern.ch` nodes. It excludes `cern.ch` nodes that can only be reached via nodes from other domains. The corresponding XQuery path expression reads as follows:

```
LET $d := ("cern.ch", "infn.it")
RETURN //node[ contains($d, domainname(service)) AND
                contains($d, domainname(ancestor::node/service))
             ]/service
```

To support such a capability a node must parse the path query, see which neighbors apply (if any) and forward a rewritten path query to them. The rewritten path query must exclude the path expression applying to the current node, essentially removing a hop from the expression. It is an open question (at least to us) how to rewrite the query correctly. Consequently, we do not consider this capability any further.

## 6.9 Containers for Centralized Virtual Node Hosting

A node link topology can be deliberately arranged and exploited by applications. For example, in an attempt to explicitly exploit topology characteristics, a virtual organization of a Grid may deliberately organize global, intermediate and local job schedulers into a tree-like topology. Correct and efficient operation of scheduling may involve queries with a neighbor selection policy that selects all child nodes and ignores all parent nodes. For the scheduling query, it is irrelevant where the nodes are running, and where and how nodes (tuples, service descriptions) are stored. What matters is that the query traverses a tree.

A *link topology* such as a star, ring, tree or graph describes the link structure among nodes, i.e. which nodes are linked with which other nodes. It is purely a logical construct. It does *not* describe where and how this link information is stored and accessed. This is defined by a *node deployment model*, which defines where and how one or more partitions of the graph are running, stored and accessed.

We argue that link topology and node deployment are distinct and orthogonal concepts, and hence a node deployment model need not correspond to a link topology at all. Consider the analogy to the WWW: The WWW is a graph of HTML pages. Vertices are established through embedded HTML hyperlinks. The graph topology is, by definition, insensitive to how and where HTML pages are physically stored and served (on which hosts, URL paths, and web server technologies). The topology remains identical, no matter whether all pages of the universe are served by a single large dynamic web server or any kind of worldwide federation of static web servers.

The simplest (and most common) deployment model has distinct nodes running on distinct hosts. However, we propose that nodes can also be concentrated in central places called node containers. A *node container* is a transparent software hosting environment that embeds one or more nodes, as depicted in Figure 6.11. The set of all nodes in the universe is partitioned over one or more node containers. A container can be a special-purpose program that *behaves as if* it were a network of nodes (*virtual hosting*). A well-known example for virtual hosting is web serving. A web server can serve millions of static or dynamic pages from an essentially infinitely large name space of URLs (nodes). To the outside world, the server is invisible, and each URL (node) can be seen as a separate service (having a name or address, HTTP network protocol, TLS security, etc). Of course, internally just one or a few processes are used to implement such virtual hosting. A general-purpose container, on the other hand, can run each node (or each request to a node) in an independent process or thread (*physical hosting*). For example, virtual and physical hosting is used in Java servlet [100] and Enterprise Java Beans technology [101].

In any case, the container is invisible to the outside world. Hosted nodes still appear and

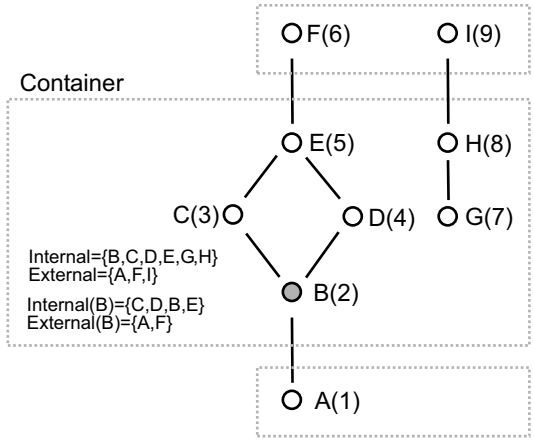


Figure 6.11: Node Containers.

behave like any other node. In our case, this means that a hosted node has a service link and description, and it supports publication, queries, etc. via the operations and network protocols advertised by the service description. The fundamental difference to classic database architectures is that in the latter there exists no deployment transparency. As an extreme example of virtual hosting, one could imagine a hypothetical relational database system that exposes each individual tuple as a network service, supporting direct network connections to the tuple to answer queries against its column values. Conceptually, we can say that every node runs within a container, even if the container holds only a single node. A remote client may ask for the dynamic creation of a virtual or physical node by means of a node factory interface. Now we are in the position to define a *node deployment model* as being a description of the set of containers physically implementing a given link topology.

Several node deployment models can be envisaged, ranging from coarse to fine grained, as well as arbitrary mixtures. For example, in a centralized deployment scenario, the entire global graph of, say $10^8$, nodes may be accessible through a single container, with all nodes (service interfaces) being handled by a single process on a single host. In a slightly less central scenario, the same graph may be partitioned among ten organizations, each with a central container as described above. Figure 6.12 depict examples along these lines. On the fully distributed end of the spectrum, each node may run on a distinct box, storing its own tuples (including neighbor descriptions) in a local registry. In all but the first case, there neither exists a single grand monolithic database nor a single owner and provider of information.

There are two primary motivations why concentrating nodes may be useful. First, for reasons including central control, reliability, continuous availability, maintainability, security, accounting and firewall restrictions on incoming connections for hosts. These reasons are important, but we do not delve into them any further. Similarly, we do not consider physical hosting any further. Instead, we focus on the second motivation, which is the potential of virtual hosting for increased performance (as opposed to increased scalability).

In any kind of P2P network, a node has a database or some kind of data source against which queries are applied. In a P2P network for service discovery, this database happens

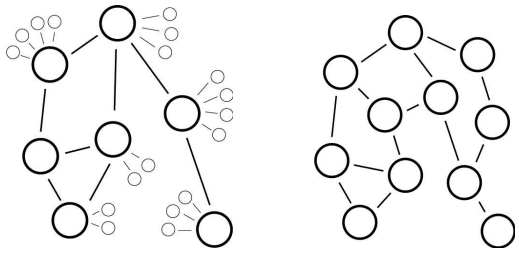

Figure 6.12: Containers partitioning a graph.

to be the publication database. Discussion in this section is applicable to any kind of P2P network, while the examples illustrate service discovery.

If many container nodes reside on the same host, in the same process and store their tuples (e.g. node service descriptions, Gnutella file indexes) in the same database, query support is potentially much more efficient. The query engine can run on "big iron". The database may fit in its entirety in a main memory buffer. Network communication between remote nodes can be replaced with local loop-back connections, inter-process communication or even direct function calls. To compute the full query result set for all container nodes, it may perhaps be sufficient to execute just one or a few batch queries against the shared database, instead of many small queries against separate databases. Intuitively it seems that the smaller nodes are, the more performance can be gained through virtual hosting. For example, consider a network with millions of small registry nodes spread all over the world, each holding just some ten tuples. Perhaps searching would be much more efficient if the nodes and their databases were just partitioned across a few, say a hundred, powerful node containers.

Consider the three example containers depicted in Figure 6.11. The central container has six internal nodes `(B,C,D,E,G,H)` and three external nodes `(A,F,I)`. External nodes belong to other containers. *Internal links* connect nodes within the container. *External links* connect internal with external nodes. A hop is said to be *logical* if it travels along an internal or external link. A hop is said to be *physical* if it travels along an external link. Intuitively it is clear that traversing an internal link is much cheaper than traversing an external link. Accordingly we propose to distinguish the separate scope parameters *logical radius* and *physical radius*. For example, a user can specify that a query should reach very far, say a logical radius of 100 hops. To ensure that this query does not burden all nodes in the universe, the user can specify that it should touch at most three containers on any given path (physical radius).

## 6.10 Query Processing with Virtual Nodes

In this section, we propose three query execution strategies. A query to a node of a container can be efficiently answered without violating the semantics of query and scope (*normal query execution*, *collecting traversal*). Even more efficiently, it can be answered by relaxing the conditions imposed by the query scope (*quick scope violating query*). Let us look at these

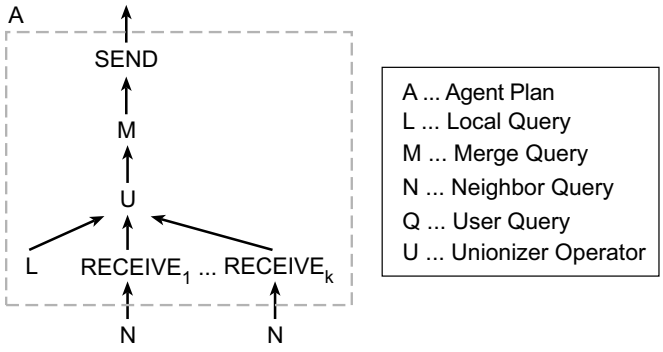


Figure 6.13: Template Execution Plan.

three strategies in more detail.

**Normal Query Execution.** Clearly a container can answer a query like any normal node via the execution plans proposed in Section 6.5. Recall the template execution plan, as depicted in Figure 6.13, and the specific plans for queries that are either recursively partitionable or not. As an optimization, network communication between internal nodes can be replaced with local loop-back connections, inter-process communication or direct function calls. For example, the Peer Database Protocol from Chapter 7 is built upon the BEEP framework. Since BEEP can be mapped to several underlying reliable transport layers (TCP is merely the default), a container can plug in an in-process transport mapping, yet continue to use the same messaging code base.

**Collecting Traversal.** To answer queries, a container can use a strategy we propose to call *collecting traversal*. Here the goal is to remove the need for any internal messaging and to run as few as possible queries against the database of shared nodes. To ensure that query semantics are fully preserved, the fact is exploited that queries in our query model are defined over a single virtual set of tuples (service descriptions), as discussed in Section 3.2. The query model allows generating this set of tuples in any arbitrary way.

The strategy works as follows. When a container node receives an external query, it takes over the work for the other internal nodes. In the first phase, it collects preparatory data. In the second phase, the query is executed. The first phase collects the internal and external nodes that are reachable from the start node. In other words, one traverses the container from the given start node, following the path that the query *would* touch. Along the way, the (keys of) internal nodes and external nodes are collected.

Consider the example from Figure 6.14. The keys of the six internal nodes are `(2,3,4, 5,7,8)` whereas the keys of the three external nodes are `(1,6,9)`. The originator sends a query to the start node `B`. The node has a database of `tuples(B)=`$\{$`1,3,4`$\}$. The internal nodes reachable from `B` are `internal(B)=`$\{$`3,4,2,5`$\}$, and the reachable external nodes are `external(B)=`$\{$`1,6`$\}$. The tuples contained in the internally reachable nodes are `tuples(internal(B)) = UNION (tuples(3), tuples(4), tuples(2), tuples(5)) =` $\{$`1,`

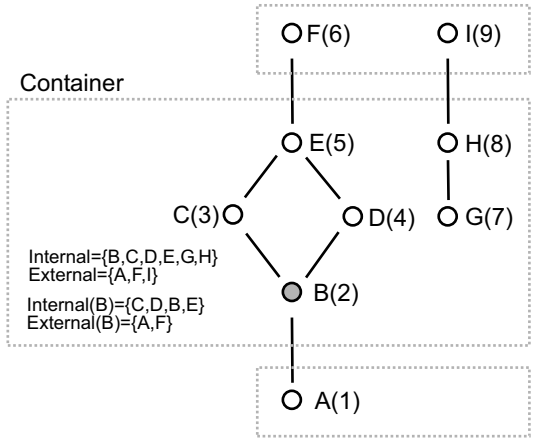


Figure 6.14: Node Containers.

2,3,4,5,6}.

According to query type, the node chooses a central or recursively partitionable execution plan, and executes it (see Section 6.5). However, the local query L is executed against the tuples of the internally reachable nodes `tuples(internal(B))` rather than against the tuples of the database `tuples(B)`. Similarly, the plan forwards the query to the nodes `external(B)` rather than to the neighbors obtained from `B`'s neighbor selection. Scope semantics are preserved by explicitly applying the relevant rules of scope parameters during node traversal (e.g. radius pruning and neighbor selection).

The net effect is that the local query `L` and the merge query `M` are batched. That is, they are applied once over a large set, instead of many times over a small set. The nodes of a container are stored in a single database (table), for example as depicted in Table 6.3. The table is not normalized for clarity of exposition. Collecting nodes is particularly fast if the neighbor selection policy is simple and the database fits into main-memory. For example, it is certainly possible to have a data structure that allows quickly traversing the database table. Further, internal messaging overhead is eliminated altogether. To summarize, if the neighbor selection policy is applied at each node, and scope parameters such as radius are observed, one can emulate normal query execution, but in a way that is more efficient.

| Node ID | Service Description | Is external? | Tuples |
|---|---|---|---|
| 1 | A | True | null |
| 2 | B | False | 3, 4 |
| 3 | C | False | 2, 5 |
| 4 | D | False | 2, 5 |
| 5 | E | False | 3, 4, 6 |
| 6 | F | True | null |

Table 6.3: Node Table of Container.

Recall the problem of *greedy radius pruning* from Section 6.8. Traversal of internal nodes

via depth-first search is inappropriate because it leads to greedy radius pruning with high probability, in particular if a container holds a large number of nodes with a complex internal topology. In practice, this means that even though a user may have specified a theoretically large enough logical radius, it is unlikely that an incoming query will ever forward beyond the current container. It is in the nature of depth-first search that it is unlikely that an external link is reached along a short internal route. Rather, it is likely that it is reached along one of the longest possible internal routes. Within a container, greedy pruning can be eliminated by traversal using breadth-first search. This ensures that the shortest path to external links is always found, despite loop detection pruning. Put another way, loop detection is conditioned to prune only paths longer than the shortest path. The net effect is that external nodes receive a meaningful logical radius scope parameter on query forward. The pseudo-code in Figure 6.15 computes the internal and external nodes of a given entry node, using breadth-first search.

**Quick Scope Violating Query.** *Normal query execution* and *collecting traversal* preserve query and scope semantics. If no query scope is given, or if it is acceptable to ignore or alter scope semantics, query execution can be optimized further using the strong technologies of centralized (relational) database architectures. For example, internal graph traversal can be eliminated altogether. The strategy works as follows. According to query type, the node chooses a central or recursively partitionable execution plan, and executes it (see Section 6.5). However, the local query `L` is executed against the union of *all* tuples of the container `(1-9)` rather than against `B`'s database `tuples(B)`. Similarly, the plan forwards the query to the external nodes selected from the union of *all* external nodes of the container `(1,6,9)` rather than to the immediate neighbors of `B`.

The net effect is that the local query `L` and the merge query `M` are batched. That is, they are applied once over a large set, instead of many times over a small set. The same holds for neighbor selection. Determining all tuples of the container requires no time at all because they are, of course, stored in the same database (table). Computing all external nodes is cheap as well. Scope-violating queries are answered using the strong technologies of centralized (relational) database architectures. Consequently, they are highly efficient, at the expense of ignoring or altering scope semantics.

For example, nodes `(7,8,9)` should never be considered, as they are not directly or indirectly connected to B. Further, it is unclear what logical radius should be assigned on query forward to external nodes. Computing the correct logical radius would essentially degrade performance down to the performance of *collecting traversal*. It appears that the least bad choice is to decrease the logical radius by one on external forward. Note that query semantics are still preserved. The query is just fed a larger than necessary set of tuples (service descriptions). In practice, this may be tolerable for a significant fraction of use cases.

## 6.11 Related Work

**Agent P2P Model.** The underlying idea of the Agent P2P model is not new. Consider for example, the email infrastructure model [33]. Typically, a single central high availability agent

```
/**
 * do:
 *   at each depth level:
 *     for each node of level:
 *       remove node from TODO list
 *       add tuples of node to INTERNAL or EXTERNAL sets
 *       append selected neighbors to TODO if not already done (loop)
 * until all nodes visited or radius limit reached
 * return INTERNAL and EXTERNAL
 * (also compute max. logicalRadius that should be used on forward to external nodes)
 */
FUNCTION collectingTraversal(startNode, logicalRadius) {

internal = {}, external = {}, done = {}, todo = {startNode}
while (size=size(todo)) > 0 and logicalRadius >= 0
   logicalRadius = logicalRadius - 1
   for i := 1 to size
        node = first element of todo
        remove first element from todo
        done = done UNION {node}
        internal = internal UNION tuples(node)
        if logicalRadius >= 0 then
          for each neighbor n IN select(neighbors(node))
             if n is internal && not contained in todo && not contained in done then
               Append n to todo
             endif
             if n is external && not (n,any radius) contained in external then
               external = external UNION {(n, logicalRadius)}
             endif
          endfor
        endif
   endfor
endwhile
Return (internal, external)
}
```

Figure 6.15: Collecting Traversal.

serves outgoing and incoming mail for originators from an entire organization. However, an email system can also be fully (or partly) decentralized such that each originator runs its own agent on its own host. This transparent flexibility contributes to the widespread adoption and tremendous success of email as an Internet "killer application". A similar example is the X.500 [13] directory architecture, which has a *Directory User Agent* (originator) querying a *Home Directory System Agent* (agent node), which is one of a collection of *Directory System Agents* (nodes).

**Loop Detection.** The X.500 protocol [13] uses a route-tracing algorithm for loop detection. This algorithm only works for queries that select on a name from a hierarchical name space that is mimicked by the link topology. In our general context, the algorithm is insufficient for loop detection because we allow, but do not assume, such a topology and namespace. The route-tracing algorithm attaches to a query the route already taken, represented by a list of node identifiers. On query forward, a node N appends its own identifier to the route. A loop is detected if the identifier of the current node N is already contained in the route. This mechanism only detects a loop if a query forwarded by a given node N eventually arrives again at the same node N. It cannot detect the more common form of loop where the same query arrives along multiple paths at a given node N, but none of the paths have so far touched N.

**Query Processing.** None of the example discovery queries from Section 3.4 can be satisfied with a lookup by key (e.g. globally unique name). This is the type of query assumed in most P2P systems such as DNS [20], Gnutella [21], Freenet [22], Tapestry [23], Chord [24] and Globe [25], leading to highly specialized *content-addressable* networks centered around the theme of distributed hash table lookup. Note further that almost no queries are exact match queries (i.e. given a flat set of attribute values find all tuples that carry exactly the same attribute values), assumed in systems such as SDS [26] and Jini [63]. Our approach is distinguished in that it not only supports all of the above query types, but it also supports queries from the rich and expressive general-purpose query languages XQuery [18] and SQL [19].

**Pipelining.** For a survey of adaptive query processing, including pipelining, see the special issue of [102]. [103] develops a general framework for producing partial results for queries involving any non-monotonic operator. The approach inserts update and delete directives into the output stream. The Tukwila [104] and Niagara projects [105] introduce data integration systems with adaptive query processing and XML query operator implementations that efficiently support pipelining. Pipelining of hash joins is discussed in [106, 107, 108]. [109] proposes a rate based pipeline scheduling algorithm that prioritizes and schedules the flow of data between pipelined operators so that the result output rate is maximized. The algorithm is also demonstrated with pipelined hash joins. Pipelining is sometimes also termed *streaming* or *non-blocking* execution.

**Neighbor Selection.** *Iterative deepening* [27] is a similar technique to *expanding ring* where an optimization is suggested that avoids reevaluating the query at nodes that have already done so in previous iterations.

Neighbor selection policies that are based on randomness and/or historical information about the result set size of prior queries are simulated and analyzed in [28].

An efficient neighbor selection policy is applicable to simple queries posed to networks in which the number of links of nodes exhibits a power law distribution (e.g. Freenet and Gnutella) [29]. Here most (but not all) matching results can be reached with few hops by selecting just a very small subset of neighbors (the neighbors that themselves have the most neighbors to the n-th radius). Note, however, that the policy is based on the assumption that not all results must be found and that all query results are equally relevant. Depending on the application context, this assumption may or may not be valid.

These related works discuss in isolation neighbor selection techniques for a particular query type, without the context of a framework for comprehensive query support.

**DNS.** Distributed databases with a hierarchical name space such as the Domain Name System (DNS) [20] can efficiently answer queries of the form *"Find an object by its full name"*. For example, the DNS can search for the IP address (e.g. `137.138.29.51`) of a given domain name (e.g. `fred.cms.cern.ch`). Because of the nature of the supported query type, these systems arrange the link topology, according to the hierarchical name space, as a tree topology, as depicted in Figure 6.16. Each node (DNS server) is responsible for tuples from a name space sub-tree such as `ch`, `cern.ch` or `cms.cern.ch`. A node may internally partition its name space and delegate responsibility for sub-trees to child nodes. A node holds a database of tuples, each of which carries a domain name and an IP address.

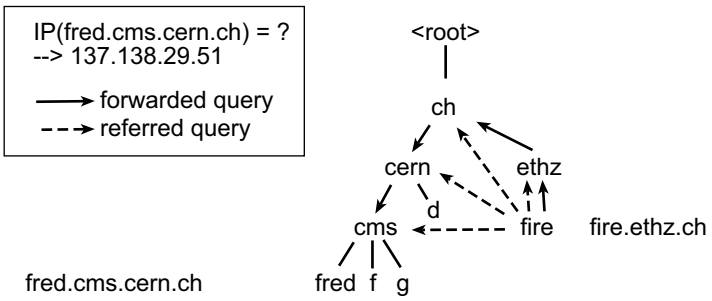


Figure 6.16: Query Flow in Domain Name System (DNS).

A query searching for the IP address of a domain name traverses the tree on the shortest path from originator (e.g. `fire.ethz.ch`) to the node containing the domain name - first up, then down. At each node, a *name resolution* policy selects the neighbor "closer" to the name than the current node, according to name space metadata. In DNS, queries are not forwarded (routed) through the topology. Instead, a node returns a *referral* message that redirects an originator to the next closer node. The originator explicitly queries the next node, is referred to yet another closer node, and so on. Nodes cache the IP address (service link result) of recently queried nodes. A cache hit directly refers a client to the responsible

node. This dramatically reduces load on nodes close to the root of the hierarchy. It also reduces the number of round trips involved for a query.

To support neighbor selection in a hierarchical name space within our UPDF framework, a node could publish to its neighbors not only its service link, but also the name space it manages. A natural candidate for the hierarchical name is the content link of a tuple. The DNS referral behavior can be implemented within UPDF by using a radius scope of zero. The same holds for the LDAP referral behavior (see below).

**X.500, LDAP and MDS.** The hierarchical distributed X.500 directory [13] works similarly to the DNS. It also supports referrals, but in addition can forward queries through the topology (*chaining* in X.500 terminology). The query language is simple (see Sections 3.4 and 3.6). Route tracing is used as a loop detection algorithm. Query scope specification can support maximum result set size limiting. It does not support radius and dynamic abort timeout as well as pipelined query execution across nodes. LDAP [14] is a simplified subset of X.500. Like DNS, it supports referrals but not query forwarding. MDS [15, 16] inherits all properties of LDAP. MDS additionally implements a simple form of query forwarding that allows for multi-level hierarchies but not for arbitrary topologies. Here neighbor selection forwards the query to LDAP servers overlapping with the query name space. The query is forwarded "as is", without loop detection[8]. Further, MDS does not support radius and dynamic abort timeout, pipelined query execution across nodes as well as direct response and metadata responses.

## 6.12 Summary

**Comparison with Related Work.** We take the first steps towards unifying the fields of database management systems and P2P computing, which so far have received considerable, but separate, attention. We extend database concepts and practice to cover P2P search. Similarly, we extend P2P concepts and practice to support powerful general-purpose query languages such as XQuery and SQL. As a result, we propose the so-called *Unified Peer-to-Peer Database Framework (UPDF)* for general-purpose query support in large heterogeneous distributed systems spanning many administrative domains. UPDF is unified in the sense that it allows to express specific applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options. The uniformity, wide applicability and reusability of our approach distinguish it from related work, which individually addresses some but not all problem areas.

Traditional distributed systems assume a particular type of topology (e.g. hierarchical as in DNS, LDAP). Existing P2P systems are built for a single application and data type and do not support queries from a general-purpose query language. For example, Gnutella, Freenet, Tapestry, Chord, Globe and DNS only support lookup by key (e.g. globally unique name).

[8]The message id which is part of every LDAP message would be unsuitable for such a purpose because an LDAP message id is not required to be universally unique. It is merely required to be different from the values of any other requests outstanding in the same (local) LDAP session of which the message is a part.

Others such as SDS, LDAP and MDS support simple special-purpose query languages, leading to special-purpose solutions unsuitable for multi-purpose service and resource discovery in large heterogeneous distributed systems spanning many administrative domains. [29] discusses in isolation neighbor selection techniques for a particular query type, without the context of a framework for comprehensive query support. LDAP and MDS do not support essential features for P2P systems such as reliable loop detection, non-hierarchical topologies, dynamic abort timeout, query pipelining across nodes as well as radius scoping. None introduce a unified P2P database framework for general-purpose query support.

**Concepts and Definitions.** The related but orthogonal concepts of (logical) link topology and (physical) node deployment model are established. A link topology describes the link structure among nodes, but it does *not* describe where and how the link information is stored and accessed. This is defined by a *node deployment model*, which defines where and how one or more partitions of the graph are running, stored and accessed.

Definitions are established, clarifying the notion of node, service, fat, thin and ultra-thin P2P networks, as well as the commonality of a P2P network and a P2P network for service discovery. A *service* exposes some functionality via interfaces to remote clients. A *node* is a service that exposes *at least* interfaces for publication and P2P queries. In a *fat* P2P network, most services are not nodes. In a *thin* or *ultra thin* P2P network, most or all services are nodes, respectively. In any kind of P2P network, nodes may publish themselves to other nodes, thereby forming a topology. In a P2P network for service discovery, services and other content providers may publish their service link and content links to nodes. Because nodes are services, also nodes may publish their service link (and content links) to other nodes, thereby forming a topology. In any kind of P2P network, a node has a database or some kind of data source against which queries are applied. In a P2P network for service discovery, this database happens to be the publication database.

**Agent P2P Model.** The agent and servent P2P models are compared. The *agent P2P model* allows fully decentralized infrastructures, yet also allows seamless integration of centralized client-server computing into an otherwise decentralized infrastructure. The *servent P2P model* is decentralized, and it does not allow for some degree of centralization.

**Loop Detection.** To reliably detect and prevent query loops, an originator attaches a different transaction identifier to each query, which is a universally unique identifier. The transaction identifier always remains identical during query forwarding over hops. A node maintains a state table of recent transaction identifiers and returns an error whenever a query is received that has already been seen.

**Response Modes.** Four techniques to return matching query results to an originator are characterized, namely Routed Response, Direct Response, Routed Metadata Response, and Direct Metadata Response. Under *Routed Response*, results are fanned back into the originator along the paths on which the query flowed outwards. Each (passive) node returns to its (active) client not only its own local results but also all remote results it receives from

neighbors. Under *Direct Response*, results are not returned by routing through intermediary nodes. Each (active) node that has local results sends them directly to the (passive) agent, which combines and hands them back to the originator. Interaction consists of two phases under *Routed Metadata Response and Direct Metadata Response*. In the first phase, routed responses or direct responses are used. However, nodes return only small metadata results. In the second phase, the originator selects which data results are relevant. The originator directly connects to the relevant data sources and asks for data results. The properties of the various response models are compared with respect to distribution and location transparency, efficiency of query support, economics, number of TCP connections at originator and agent, latency, caching and trust delegation to unknown parties.

**Query Processing.** From the perspective of query processing, a P2P database system has the same properties as a distributed database system, in a recursive structure. In a distributed database system, there exists a single local database and zero or more neighbors. A classic centralized database system is a special case where there exists a single local database and zero neighbors. Hence, we propose to organize the P2P query engine like a general distributed query engine. Any query against any kind of database system can be answered within the proposed framework. A P2P network can be efficient in answering queries that are recursively partitionable. A query is *recursively partitionable* if the very same execution plan *can* be recursively applied at every node in the P2P topology. The recursive parallel spread of load implied by a recursively partitionable query is the basis of the much-cited massive P2P scalability potential. The definition of simple, medium and complex queries is clarified. A query is *simple* if it is recursively partitionable using the UNION operator for unionizing and the IDENTITY operator for merging. A query is a *medium* query if it is not simple, but it is recursively partitionable. A query is *complex* if it is not recursively partitionable.

**Pipelining.** Often an originator would be happy to already do useful work with one or a few *early tuple results*, as long as they arrive quickly and reliably. Results that arrive later can be handled later, or are ignored anyway. The semantics of certain operators allows them to support pipelining, while others do not. Whether an agent exhibits a short or long latency to deliver to the originator the first result from the result set depends on the query operators in use, which in turn depend on the given query. In other words, for some query types the originator has the potential to immediately start piping in results (at moderate performance rate), while for other query types it must wait for a long time until the first result becomes available (the full result set arrives almost at once, however). Simple queries do support pipelining. Medium queries may or may not support pipelining, whereas complex queries typically do not support pipelining.

**Static Loop Timeout and Dynamic Abort Timeout.** Clearly there comes a time when a user is no longer interested in query results, no matter whether any more results might be available. The query roaming the network and its response traffic should fade away after some time. The value of a *static timeout* remains unchanged across hops. In contrast, it is intended that the value of a *dynamic timeout* be decreased at each hop. Nodes

further away from the originator may time out earlier than nodes closer to the originator. Non-pipelined delivery with a static abort timeout is highly unreliable due to the so-called simultaneous abort timeout problem. To address the problem, we propose *dynamic abort timeouts* using as policy *exponential decay with halving*. This ensures that a maximum of results can be delivered reliably within the time frame desired by a user. A dynamic timeout is unsuitable to be used as *loop timeout*, due to the non-simultaneous loop timeout problem. A loop timeout must be static.

**Query Scope.** A query scope is used to navigate and prune the link topology and filter on attributes of the deployment model. Conceptually, the scope is the input fed to the query. One can indirectly specify a scope based on neighbor selection, timeout, radius and result set properties. A node forwards a query to the set of nodes obtained from *neighbor selection*. The best neighbor selection policy to adopt depends on the context of the query and the topology. For example, a query may only select neighbors that meet minimum requirements in terms of latency and bandwidth. Using neighbor selection explicit topology characteristics can be exploited in query guidance. For flexibility and expressiveness, we propose to allow the user to specify the selection policy. In addition to the normal query, the user defines a *neighbor selection query* (XQuery) that takes the tuple set of the current node as input and returns a subset that indicates the nodes selected for forwarding. A wide range of policies can be implemented in this manner. The neighbor selection policy can draw from the rich set of information contained in the tuples published to the node. The *radius* of a query is the maximum number of hops a query is allowed to travel on any given path. The radius can be used to indirectly limit result set size. In addition, it helps to limit latency and bandwidth consumption and to guard against runaway queries with infinite lifetime. Loop detection and unpredictable timing in distributed systems can lead to *greedy radius pruning*.

**Containers for Centralized Virtual Node Hosting.** Link topology and node deployment are distinct and orthogonal concepts, and hence a node deployment model need not correspond to a link topology at all. The simplest (and most common) deployment model has distinct nodes running on distinct hosts. A *node container* is a transparent software-hosting environment that embeds one or more nodes. The set of all nodes in the universe is partitioned over one or more node containers. A container can be a special-purpose program that *behaves as if* it were a network of nodes (*virtual hosting*). A well-known example for virtual hosting is web serving. Hosted nodes still appear and behave like any other node. In our case, this means that a hosted node has a service link and description, and it supports publication, queries, etc. via the operations and network protocols advertised by the service description. Node deployment models range from centralized to fully distributed. Virtual hosting has the potential for increased performance (as opposed to increased scalability). For example, consider a network with millions of small registry nodes spread all over the world, each holding just some ten tuples. Perhaps searching would be much more efficient if the nodes and their databases were just partitioned across a few, say a hundred, powerful node containers.

*Internal links* connect nodes within a container. *External links* connect internal with external nodes. The separate scope parameters *logical radius* and *physical radius* are dis-

tinguished. A query to a container node can be efficiently answered without violating the semantics of query and scope (*normal query execution*, *collecting traversal*). Even more efficiently, it can be answered by relaxing the conditions imposed by the query scope (*quick scope violating query*).

The goal of *collecting traversal* is to remove the need for any internal messaging and to run as few as possible queries against the database of shared nodes. To ensure that query semantics are fully preserved, the fact is exploited that queries in our query model are defined over a single virtual set of tuples. The query model allows generating this set of tuples in any arbitrary way. The first phase collects the internal and external nodes that are reachable from the start node. The local query is executed against the tuples of the internally reachable nodes. The query is forwarded to the reachable external nodes. Scope semantics are preserved by explicitly applying the relevant rules of scope parameters during node traversal. Traversal of internal nodes via depth-first search is inappropriate because it leads to greedy radius pruning with high probability. Breadth-first search should be used instead.

If no query scope is given, or if it is acceptable to ignore or alter scope semantics, a query can be answered with the *quick scope violating query* strategy, using the strong technologies of centralized (relational) database architectures. Internal graph traversal is eliminated altogether. The local query, the merge query and neighbor selection are applied once over a large set, instead of many times over a small set.

# Chapter 7

# A Unified Peer-to-Peer Database Protocol

## 7.1 Introduction

Recall that Chapter 4 designed the *hyper registry*, which is a centralized database (node, peer) for discovery of dynamic distributed content. Section 5.2 formulated the corresponding `XQuery` interface. Chapter 6 devised the Unified Peer-to-Peer Database Framework (UPDF) that is unified in the sense that it allows to express specific applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options. In this chapter, we propose a messaging model and network protocol that supports the UPDF framework, the hyper registry and the `XQuery` interface.

The design of a messaging model and network protocol for large distributed systems strongly influences system properties such as scalability, efficiency, interoperability, extensibility, reliability, and, of course, limitations in applicability. For example, if the messaging model does not support pipelining, a result set has to be delivered with long latency in a single large batch, even though the query type might allow for pipelining. Clearly this may render a system inappropriate for some interactive or quasi real-time applications. The key problem is:

- *What messaging and communication model as well as network protocol uniformly supports P2P database queries for a wide range of database architectures and response models such that the stringent demands of ubiquitous Internet discovery infrastructures in terms of scalability, efficiency, interoperability, extensibility and reliability can be met? In particular, how can one allow for high concurrency, low latency as well as early and/or partial result set retrieval? How can one encourage resource consumption and flow control on a per query basis?*

In this chapter, these problems are addressed by developing a suitable messaging, communication and network protocol model, collectively termed *Peer Database Protocol (PDP)*. PDP has a number of key properties. It is applicable to any node topology (e.g. centralized, distributed or P2P) and to multiple P2P response modes (routed response and direct response, both with and without metadata modes). To support loosely coupled autonomous Internet infrastructures, the model is connection-oriented (ordered, reliable, congestion sensitive) and message-oriented (loosely coupled, operating on structured data). For efficiency, it is stateful at the protocol level, with a transaction consisting of one or more discrete message

exchanges related to the same query. It allows for low latency, pipelining, early and/or partial result set retrieval due to synchronous pull, and result set delivery in one or more variable sized batches. It is efficient, due to asynchronous push with delivery of multiple results per batch. It provides resource consumption and flow control and on a per query basis, due to the use of a distinct channel per transaction. It is scalable, due to application multiplexing, which allows for very high query concurrency and very low latency, even in the presence of secure TCP connections. To encourage interoperability and extensibility it is fully based on Internet Engineering Task Force (IETF) standards, for example in terms of asynchrony, encoding, framing, authentication, privacy and reporting. We believe that the Peer Database Protocol is suitable to meet the stringent demands of ubiquitous Internet infrastructures.

Starting from high-level, the model is developed in increasing levels of detail. Message types and their semantics are introduced and specified in detail. State transitions related to message handling are specified. Message representations, communication model and network protocol are specified. The minimum state information a node must maintain for correct operation is introduced.

## 7.2 Originator and Node

Recall from Section 6.2 that in the agent P2P model an *originator* hands a query to an *agent node*, which applies the query to its local database and also forwards the query to its neighbor *nodes*. To be able to query each other, all nodes must support the same protocol `P`. From the functional perspective, there is no difference between an originator querying an agent, and a node querying another node. Consequently, the protocol `P` can also be used by an originator to query an agent.

However, the relationship between originator and agent may take any form. For example, an originator may embed its agent in the same process. The originator may just as well choose a remote node as agent, for reasons including central control, reliability, continuous availability, maintainability, security, accounting and firewall restrictions on incoming connections for originator hosts. A simple HTML GUI may be sufficient to originate queries that are sent to an organization's agent node. For flexibility, we do not mandate any particular communication protocol between originator and agent. A node is free to implement any number of additional protocols for communication with originators and offer them via additional service interfaces.

For example, it may implement a simple and straightforward HTTP based query protocol for web browser based originators. This way, originators have the option to alternatively use a simpler mechanism than the (non-trivial) protocol `P`. Consequently, one may, but need not, distinguish communication protocols between a) originator and agent node and b) between nodes. In the remainder of this chapter, no such distinction is made. We assume for simplicity that, if originators are remote, they also speak the protocol `P` - the Peer Database Protocol.

## 7.3 High-Level Messaging Model

The high-level messaging model employs four request messages (QUERY, RECEIVE, INVITE, CLOSE) and a response message (SEND). A *transaction* is a sequence of one or more message exchanges between two peers (nodes) for a given query. An example transaction is a QUERY-RECEIVE-SEND-RECEIVE-SEND-CLOSE sequence[1]. This non-trivial transaction model is in contrast to a simpler model where a transaction consists of a single request-response exchange (e.g. HTTP). The former model is stateful whereas the latter is stateless. A peer can concurrently handle multiple independent transactions. A transaction is identified by a transaction identifier. Every message of a given transaction carries the same transaction identifier. The messages have the following semantics:

- **QUERY.** A QUERY message is forwarded along hops through the topology. The message contains the query itself as well as a universally unique transaction identifier (UUID) for the query (see Section 6.3). Every other message below refers to a prior QUERY message by carrying the same transaction identifier. The QUERY message also contains scope hints. It may optionally also contain a hint indicating what response mode should be used (`routed` or `direct`). Under direct response, also the service link or description of the agent must be included, so that nodes with matches can invite the agent to retrieve the result set. Optionally, the identity of the originator may be included to allow for authorization decisions where applicable. A node accepting a QUERY message returns immediately without any results. Results are explicitly requested via a subsequent RECEIVE message.

- **RECEIVE.** A RECEIVE message is used by a client to request query results from another node. It requests the node to SEND a batch of at least `N` and at most `M` results from the (remainder of the) result set. This corresponds to the `next()` method of an iterator (operator). We have $1 \leq N \leq M$. For example, a low latency use case can use `N=1, M=10` to indicate that at least one and at most ten results should be delivered by the next batch. `N=M=infinity` indicates that all remaining results should be send in a single large batch.

- **SEND.** When a node accepts a RECEIVE message, it responds with a SEND message, containing a batch with the requested results from the (remainder of the) result set. A client can successively issue multiple RECEIVE messages until the result set is exhausted. A client need not retrieve all results from the entire result set. For example, after the first batch of 10 results it may issue a CLOSE request.

- **CLOSE.** A client may issue a CLOSE request to inform a node that the remaining results (if any) are no longer needed and can safely be discarded.

- **INVITE.** INVITE messages only apply to direct response mode. A node forwards the query without ever waiting for remote result sets. It only applies the query to its local

[1]This notion is entirely unrelated to the notion used in database systems where a transaction is an atomic unit of database access which is either completely executed or not executed at all [93].

database. If the local result set is not empty, the node directly contacts the agent with an INVITE message to solicit a RECEIVE message. Interaction then proceeds with the normal RECEIVE-SEND-CLOSE pattern, either in a synchronous or asynchronous manner (see below).

- **Synchronous (pull) vs. asynchronous (push) RECEIVE.** A RECEIVE request contains a parameter that asks to deliver SEND messages in either synchronous (pull) or asynchronous (push) mode. In synchronous mode a single RECEIVE request must precede every single SEND response. An example sequence is RECEIVE-SEND-RECEIVE-SEND. In asynchronous mode a single RECEIVE request asks for a sequence of successive SEND responses. A client need not explicitly request more results, as they are automatically pushed in a sequence of zero or more SENDs. An example sequence is RECEIVE-SEND-SEND-SEND.

As a practical example, assume a query that yields 100 results, and an originator that consumes a batch of `N=M=50` results per SEND. The messaging model is exemplified by the corresponding message flows from client to server ("`-->`") and back ("`<--`") depicted in Table 7.1.

| **Routed Synchronous Response** | **Routed Asynchronous Response** | **Direct Synchronous Response** | **Direct Asynchronous Response** |
|---|---|---|---|
| `--> QUERY`<br>`--> RECEIVE`<br>`<-- SEND`<br>`--> RECEIVE`<br>`<-- SEND`<br>`--> CLOSE` | `--> QUERY`<br>`--> RECEIVE`<br>`<-- SEND`<br>`<-- SEND`<br>`--> CLOSE` | `--> QUERY`<br><br>`<-- INVITE`<br>`--> RECEIVE`<br>`<-- SEND`<br>`--> RECEIVE`<br>`<-- SEND`<br>`--> CLOSE` | `--> QUERY`<br><br>`<-- INVITE`<br>`--> RECEIVE`<br>`<-- SEND`<br>`<-- SEND`<br>`--> CLOSE` |

Table 7.1: High-Level Message Flow.

Concerning the number of results, the detailed semantics of SEND and INVITE remain to be specified. Further, propagation semantics of CLOSE need to be stated. The revised specifications read as follows:

- **SEND.** When a node accepts a RECEIVE message, it responds with a SEND message, containing a batch with `P` results from the (remainder of the) result set. A client can successively issue multiple RECEIVE messages until the result set is exhausted. A client need not retrieve all results from the entire result set. For example, after the first batch of 10 results it may issue a CLOSE request. We have $\texttt{P} \leq \texttt{M}$. We may, but need not, have $\texttt{N} \leq \texttt{P}$. For example, less than `N` results may be delivered when the entire query result set is exhausted, or if the node decides to override and decrease `N` (e.g. for reasons including resource consumption control).

A SEND message also contains the number `R` of remaining results currently available for immediate non-blocking delivery with the next SENDs (`nonBlockingResultsAvailable`). Usually `R` is greater than zero. `R=0` can indicate that remote nodes have not yet delivered results necessary to return more than zero results with the next SEND. `R=-1` indicates that the batch contains the last results as the result set is exhausted. No more RECEIVE messages must be issued after that point. `R=-2` indicates that the number is unknown.

A SEND message also contains the current estimate `Q` of the remaining total result set size, irrespective of blocking (`estimatedResultsAvailable`). The actual number of results that can (later) be delivered may be larger. It should not be smaller, except if other nodes fail to deliver their suggested results. Usually `Q` is greater or equal to zero. `R=-1` implies `Q=-1`, indicating that the result set is definitely exhausted. `Q=-2` indicates that the number is unknown.

In synchronous mode a single RECEIVE request must precede every single SEND response. In asynchronous mode a single RECEIVE request asks for a sequence of successive SEND responses, each of which contains a batch with `P` results from the (remainder of the) result set.

- **INVITE.** Under direct response, a node forwards the query without ever waiting for remote result sets. It only applies the query to its local database. If the local result set is not empty, the node directly contacts the agent with an INVITE message to solicit a RECEIVE message. Interaction then proceeds with the normal RECEIVE-SEND-CLOSE pattern, either in a synchronous or asynchronous manner. An INVITE message also contains the number `R` of results currently available for immediate non-blocking delivery (`nonBlockingResultsAvailable`). `R` must be greater or equal to zero. The message also contains the current estimate `Q` of the remaining total result set size, irrespective of blocking (`estimatedResultsAvailable`). `Q` must be greater than zero.

- **CLOSE.** A client may issue a CLOSE message to inform a node that the remaining results (if any) are no longer needed and can safely be discarded. A CLOSE message responds immediately with an acknowledgement. At the same time, the node asynchronously forwards the CLOSE to neighbors involved in result set delivery, which in turn forward the CLOSE to their neighbors, and so on. Being informed of a CLOSE allows a node to release resources as early as possible. Strictly speaking, a client need not issue a CLOSE, and a node need not forward further a CLOSE, because a query eventually times out anyway. Even though this is considered misbehavior, a node must continue to operate reliably under such conditions.

A node may periodically discover other peers and announce its presence. Node discovery uses a QUERY that selects all tuples with node service descriptions. For presence announcement, a node additionally includes its service description as optional QUERY data. Explicit PING/PONG messages as used in Gnutella [21] are unnecessary.

Clearly a RECEIVE request may cause cascading RECEIVEs through the P2P node topology, followed by cascading SEND responses backwards. In the worst case every RECEIVE cascades through a large number of node hops, incurring prohibitive latencies. This

highlights the importance of (appropriately sized) batched delivery, which greatly reduces the number of hops incurred by a single RECEIVE. Also, note that the I/O of a node need not be driven strictly by client demand. For example, in an attempt to reduce latency, a node accepting a QUERY may already prefetch query results from its neighbors even though it has not yet seen the corresponding RECEIVE request from its client.

**State Transitions.** A node maintains a state table. For each query at least the transaction identifier (abbreviated `tid`), abort timeout, loop timeout and an open/closed state flag are kept. An example state table reads as follows:

| Tid | Abort Timeout | Loop Timeout | State |
|---|---|---|---|
| 100 | 20 | 30 | Closed |
| 200 | 50 | 60 | Open |

A query is *known* to a node if the state table already holds a transaction identifier equal to the transaction identifier of the query. Otherwise, it is said to be *unknown*. A known query can be in two states: *open* or *closed*. Let us discuss the state transitions from *unknown* to *open* to *closed* and back to *unknown* state.

- **Open.** When an *unknown* query arrives with a QUERY message, it moves into *open* state. When a query moves into *open* state, it becomes known and is forwarded to the neighbors obtained from neighbor selection.
- **Closed.** A query moves from *open* into *closed* state when its abort timeout has been reached, or if the result set is exhausted by the final SEND, or if a client issues a CLOSE to indicate that it is no longer interested in the (remainder of the) result set, or if one of several errors occur. Under direct response, a query *also* moves from *open* into *closed* state if the query produces no local results, or if it does produce local query results but the INVITE request is not accepted by the agent.

  In any case, when a query moves into *closed* state, a CLOSE request is asynchronously forwarded to all dependents in order to inform them as well. A node depends on a set of other nodes (*dependents*) that are involved in result set delivery. Under Routed Response, the dependants are the nodes obtained from neighbor selection. Under Direct Response, the dependents of an agent are the nodes from which the agent has accepted an INVITE message, whereas all other nodes have no dependents.
- **Unknown.** A query moves from *closed* state into *unknown* state when its loop timeout has been reached. In other words, the query is deleted from the state table.
- **Message Acceptance and Rejection.** A QUERY request is accepted if the query is *unknown*. If an already known QUERY arrives, this usually indicates loop detection. The message is rejected with an error (e.g. "transaction identifier already in use"). When a message other than QUERY arrives that has an unknown transaction identifier, it is rejected with an error (e.g. "transaction identifier unknown"). RECEIVE, SEND, CLOSE and INVITE messages are accepted for a query in *open* state. No message for a query in *closed* state is accepted; the response to a message is always an error (e.g. "transaction identifier already closed").

The state transitions are summarized in Figure 7.1.

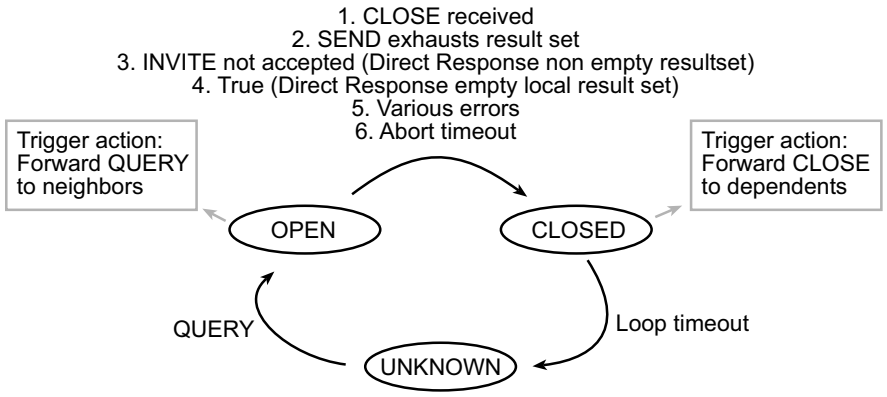


Figure 7.1: Node State Transitions.

## 7.4 Concrete Messages

The high-level messaging model proposed so far omits many details. For example, any realistic messaging model must deal with acknowledgment and error messages. In a straightforward manner, the model is now mapped down to the abstract messaging model of the BEEP application level network protocol framework [35, 36]. BEEP is an IETF standard designed for connection-oriented (ordered, reliable, congestion sensitive), message-oriented (loosely coupled, structured data), asynchronous (peer-to-peer, allowing client-server) communications. The framework defines one *request* message class (MSG) and four *response* messages classes (RPY, ERR, ANS, NULL). Discrete messages belong to well-defined message exchange patterns. For example, the pattern of synchronous exchanges (one-to-one, pull) is supported as well as the pattern of asynchronous exchanges (one-to-many, push). The response to a MSG message may be an *error* (ERR), a *reply* (RPY) or a sequence of zero or more *answers* (ANS), followed by a *null terminator* message (NULL). The exchange patterns are summarized as follows:

```
MSG --> RPY | (ANS [0..N], NULL) | ERR
```

The BEEP framework explicitly expects each message class to be extended by applications as necessary. Accordingly, the messages QUERY, RECEIVE, SEND, CLOSE, INVITE are refined, yielding three request MSG types (MSG QUERY, MSG RECEIVE, MSG INVITE), two reply message types (RPY SEND, RPY OK), one answer message type (ANS SEND), and the ERR error type. The RPY OK and ERR message type are introduced because any realistic messaging model must deal with acknowledgments and errors. The following message exchanges are permitted:

```
MSG QUERY   --> RPY OK | ERR
MSG RECEIVE --> RPY SEND | (ANS SEND [0:N], NULL) | ERR
```

```
MSG INVITE  --> RPY OK | ERR
MSG CLOSE   --> RPY OK | ERR
```

Using the refined model, the message flows depicted in Table 7.2 can occur in the running example.

| **Routed Synchronous Response** | **Routed Asynchronous Response** | **Direct Synchronous Response** | **Direct Asynchronous Response** |
|---|---|---|---|
| --> MSG QUERY | --> MSG QUERY | --> MSG QUERY | --> MSG QUERY |
| <-- RPY OK | <-- RPY OK | <-- RPY OK | <-- RPY OK |
| --> MSG RECEIVE | --> MSG RECEIVE | | |
| <-- RPY SEND | <-- ANS SEND | <-- MSG INVITE | <-- MSG INVITE |
| --> MSG RECEIVE | <-- ANS SEND | --> RPY OK | --> RPY OK |
| <-- RPY SEND | <-- NULL | --> MSG RECEIVE | --> MSG RECEIVE |
| --> MSG CLOSE | --> MSG CLOSE | <-- RPY SEND | <-- ANS SEND |
| <-- RPY OK | <-- RPY OK | --> MSG RECEIVE | <-- ANS SEND |
| | | <-- RPY SEND | <-- NULL |
| | | --> MSG CLOSE | --> MSG CLOSE |
| | | <-- RPY OK | <-- RPY OK |

Table 7.2: Refined Message Flow.

Message types and their parameters can be mapped to multiple representations. For simplicity and flexibility, we use straightforward XML [11] representations. Without loss of generality, example query expressions (e.g. user query, merge query and neighbor selection query) are given in the XQuery language [18], as detailed in Chapter 6. SQL [19], LDAP [14] or other query languages could be used as well. Consider the following example messages:

```
<MSG_QUERY transactionID = "12345">
  <query>
        <userquery> RETURN /tupleset/tuple </userquery>
        <mergequery unionizer="UNION"> RETURN /tupleset/tuple </mergequery>
  </query>
  <scope loopTimeout = "2000000000000" abortTimeout = "1000000000000"
         logicalRadius = "7" physicalRadius = "4"
         maxResults = "100"  maxResultsBytes = "100000">
         <neighborSelectionQuery>         <!-- implements broadcasting -->
            RETURN /tupleset/tuple[@type="service"
              AND content/service/interface[@type="Consumer-1.0"]
              AND content/service/interface[@type="XQuery-1.0"]]
         </neighborSelectionQuery>
  </scope>
  <options>
         <responseMode> direct </responseMode>
         <agent>
            <tuple link="http://fred.example.com/getServiceDescription"
                   type="service" TS1="20" TC="25" TS2="30" TS3="40">
              <content>
                <service> service description of agent goes here </service>
```

```
            </content>
          </tuple>
        </agent>
        <originator> fred@example.com </originator>
  </options>
</MSG_QUERY>


<MSG_RECEIVE transactionID = "12345">
  <mode minResults = "1" maxResults = "10">
     synchronous
  </mode>
</MSG_RECEIVE>


<RPY_SEND transactionID = "12345">
  <data nonBlockingResultsAvailable = "-1" estimatedResultsAvailable = "-1">
    <tupleset TS4="100">
       <tuple link="http://sched.infn.it:8080/pub/getServiceDescription"
              type="service" ctx="child" TS1="20" TC="25" TS2="30" TS3="40">
          <content>
             <service> service description B goes here </service>
          </content>
       </tuple>
       <tuple link="http://repcat.cern.ch/pub/getStatistics"
              type="repcatStats" TS1="60" TC="65" TS2="70" TS3="80">
          <content>
              <repcatStats host="repcat.cern.ch" avgHitsPerMin="1000">
                <dbsize countLFNs="100000" countPFNs="100000000"/>
              </repcatStats>
          </content>
       </tuple>
    </tupleset>
  </data>
</RPY_SEND>


<ANS_SEND transactionID = "12345">
    structure is identical to RPY_SEND ...
</ANS_SEND>


<MSG_INVITE transactionID = "12345">
  <avail nonBlockingResultsAvailable="50" estimatedResultsAvailable="100"/>
</MSG_INVITE>


<MSG_CLOSE transactionID = "12345" code="555"> maximum idle time exceeded
</MSG_CLOSE>

<RPY_OK transactionID = "12345"/>


<ERR transactionID = "12345" code="550"> transaction identifier unknown </ERR>
```

## 7.5 Communication Model and Network Protocol

The previous sections proposed a messaging model in increasing levels of detail. In this section, the communications between two peers (e.g. originator and agent node, or node and another node) are described by a *communication model*. The model operates on abstract entities such as sessions, channels, messages and frames. The communication model is later explicitly mapped down to a network protocol. A *network protocol* describes how abstract entities such as sessions and channels map to physical entities such as TCP connections. Further, a network protocol spells out how to handle asynchrony (handling independent exchanges), encoding (representing messages), framing (delimiting messages), authentication (verifying user identities), privacy (protecting against third-party interception) and reporting (conveying status information such as errors).

**Communication Model.** We adopt the communication model of BEEP because it well fits the requirements of our (non-trivial) messaging model. The model has the following properties.

- **Session, Channel, Message and Frame.** Two peers establish a *session* for communication. Within a session, one or more *channels* can be established. A channel carries zero or more *messages*. A message can have arbitrary length and content. A message is segmented into one or more *frames* of variable length. A session is established by an initiator for communication with a listener. Within a session, the peer that awaits new channels is acting in the server role, and the other peer, which establishes a channel to the server, is acting in the client role. In P2P style, both initiator and listener may (but need not) act as client and server at the same time.

- **Intra-channel.** Within a channel, all messages are processed in serial order. The server must generate responses in the same order as corresponding request messages are received. One or more request messages may be issued without waiting to receive the corresponding responses. That is, a channel provides *pipelining*. To this end, each request message carries an integer identifier that is unique within the channel. Responses to the message carry the same identifier.

- **Inter-channel.** Channels are isolated from each other, and therefore handle asynchrony/multiplexing. A channel cannot "see" or interfere with messages from other channels. There are no constraints on the processing order for different channels. In other words, inter-channel messages may be unordered.

- **Flow control.** In all likelihood, the concurrent channels of a session are carried over the same physical network cable. Consequently, flow control policy issues arise: If more than one channel offers a frame to send, which frame should be chosen? What is a good size for a frame? A peer may implement any policy it sees fit. For example, it may attempt to prevent starvation and encourage fairness. A slow channel should not be able to monopolize the session. Large messages may be segmented into multiple frames, and different channels may be served in round-robin fashion or according to priorities.

The larger the system load, the more important is flow control for reliable and predictable operation. Consider the analogy between P2P nodes and physical IP routers, which form nodes in a multi hop packet switching network. Obviously it is desirable to be able to control the Quality of Service policies, I/O scheduling and queues of IP routers.

Recall that a peer can concurrently handle multiple independent transactions. A transaction is a sequence of one or more message exchanges between two peers for a given query. An example transaction is a QUERY-RECEIVE-SEND-RECEIVE-SEND-CLOSE sequence. This non-trivial transaction model is in contrast to a simpler model where a transaction consists of a single request-response exchange. The simple or non-trivial use of transactions determines the context in which channels are used. In the simple model, it may be appealing to share a channel for many unrelated transactions. In our non-trivial model, *one channel per distinct transaction* is used to isolate communication referring to different queries and to ensure that messages of a transaction are processed in serial order. This also provides for resource consumption and flow control on a per query basis.

Let us extend the running example to cover two concurrent interleaved queries `Q1` and `Q2`. `Q1` is dealt with on channel 1, whereas `Q2` is dealt with on channel 2. Even though `Q1` is issued before `Q2`, `Q2` is answered earlier, for any of a variety of reasons. `Q2` may be simpler than `Q1`. Nodes involved in answering `Q1` may be busy or down. The bandwidth of network paths may strongly vary. The flow control policy of a node may degrade the priority of `Q1`. In the example, the multiplexed message flows depicted in Table 7.3 can occur (notation is `channelNumber : message`). Only routed response modes are shown. Direct response modes are analogous.

| **Routed Synchronous Response** | **Routed Asynchronous Response** |
|---|---|
| `--> 1: MSG QUERY`<br>`<-- 1: RPY OK`<br>`--> 2: MSG QUERY`<br>`<-- 2: RPY OK`<br>`--> 1: MSG RECEIVE`<br>`--> 2: MSG RECEIVE`<br>`<-- 2: RPY SEND`<br>`--> 2: MSG RECEIVE`<br>`<-- 2: RPY SEND`<br>`--> 2: MSG CLOSE`<br>`<-- 1: RPY SEND`<br>`<-- 2: RPY OK`<br>`--> 1: MSG CLOSE`<br>`<-- 1: RPY OK` | `--> 1: MSG QUERY`<br>`<-- 1: RPY OK`<br>`--> 2: MSG QUERY`<br>`<-- 2: RPY OK`<br>`--> 1: MSG RECEIVE`<br>`--> 2: MSG RECEIVE`<br>`<-- 2: ANS SEND`<br>`<-- 1: ANS SEND`<br>`<-- 2: ANS SEND`<br>`<-- 2: NULL`<br>`--> 2: MSG CLOSE`<br>`<-- 1: ANS SEND`<br>`<-- 2: RPY OK`<br>`<-- 1: NULL`<br>`--> 1: MSG CLOSE`<br>`<-- 1: RPY OK` |

Table 7.3: Multiplexed Message Flows.

**Network Protocol.** A messaging and communication model does not specify how to handle asynchrony (handling independent exchanges), encoding (representing messages), framing (delimiting messages), authentication (verifying user identities), privacy (protecting against third-party interception) and reporting (conveying status information such as errors). In this section, we specify details on these issues. Further, a communication model does not operate on physical entities such as TCP connections and packets, but rather on abstract entities such as sessions, channels, messages and frames. The concepts of session and channel can be mapped to concrete physical entities in several ways:

- **Session.** There are two options: one session per originator, or one session per initiator (neighbor node). Under the former option, a new session between two peers is established whenever a query from a previously unknown originator is accepted. The session is shared by all queries from the given originator. This option does not scale well in the presence of many concurrent originators. In contrast, under the latter option, a new session is established whenever a new node publishes itself as neighbor (more lazy: when the first query exchange with a neighbor happens). Irrespective of how many originators issue queries, the session persists until a node leaves the network. Since neighbors do not join and leave very frequently, this option involves less latency because session establishment occurs less often.

- **Channel.** As has been noted, our model uses one channel per distinct transaction identifier in order to isolate communications referring to different queries and to ensure that messages of a transaction are processed in serial order. In other words, there exists one channel per query. There are two options to map TCP connections: one TCP connection per channel (*TCP multiplexing*, *TM*) and one TCP connection per session (*application multiplexing, AM*).

  TM is easy to implement because multiplexing is directly supported by the TCP stack, which natively handles multiple concurrent TCP connections. An application need not bother how to implement multiplexing and flow control, but it also has few means to control it. For simplicity, almost all network protocols use TM. Under AM, all channels of a session share a single TCP connection. Typically, each channel has an associated memory buffer. AM is much more complex because multiplexing must be supported on top of the TCP stack, at the application level. Note, however, that the complexities involved are well taken care of by existing commodity software frameworks such as `beepcore` [110]. TM has the distinct disadvantage of being much less efficient in the presence of high frequency channel creation. With new queries (channels) arriving at high frequency, TM encounters serious latency limitations due to the very expensive nature of secure (and even insecure) TCP connection setup. Even if TCP connections are kept alive, pooled and reused, at least `N*neighbors` TCP connections are needed under broadcast to handle `N` concurrent queries. While this solution may be perfectly adequate for small special-purpose networks, it clearly does not scale well to the stringent demands of a ubiquitous Internet infrastructure such as service discovery. This is precisely the demanding scenario AM is designed for: Channel establishment only requires a single message exchange over an already existing TCP connection. If channels

are pooled and reused, channel establishment is a null operation and does not involve any network communication exchange.

We assume that at any given moment in time, a node faces many queries, fewer originators and even fewer initiators (neighbors). In a widely deployed system, a node must expect to face hundreds to many thousands of concurrent queries, hundreds to thousands of concurrent originators, and tens to hundreds of neighbors. For example, many queries are periodically reissued with substantial frequency. Many queries are in open state but have long interactive latencies between consecutive RECEIVE-SEND exchanges. Many queries are in open state waiting for abort timeout, because client programs tend to be lazy and forget to issue a CLOSE. Many queries are in closed state waiting for loop timeout.

It appears unnecessary and inefficient to setup up a new session for each originator. Hence, one session per initiator (neighbor node) is used. For simplicity and easy authorization, we actually use two sessions per initiator (neighbor node). One session is used for traffic related to incoming queries (queries posed to the node), the other for traffic related to outgoing queries (queries the node poses to another node). A node with `N` neighbors has `N` incoming and `N` outgoing sessions. In a successful P2P network, queries do indeed arrive at high frequencies. Session establishment may be heavyweight, but channel establishment must be lightweight. Hence, application multiplexing is chosen. A node with `N` neighbors has `N` incoming and `N` outgoing TCP connections.

Any network protocol must deal with a set of common problems. The BEEP framework was introduced to avoid the need to reinvent solutions to common problems. BEEP is an IETF standard designed for connection-oriented (ordered, reliable, congestion sensitive), message-oriented (loosely coupled, structured data), asynchronous (peer-to-peer, allowing client-server) communications. We propose to adopt the framework because it integrates existing best-of-breed standards. BEEP uses channels for *asynchrony* (handling independent exchanges). The transport mapping to TCP uses application multiplexing (one TCP connection per session) with sliding windows [36]. More precisely, each channel has a sliding window that indicates the number of payload octets that a peer may transmit before receiving further permission to transmit. MIME [67] with a default of `text/xml` is used for *encoding* (representing messages). Octet counting with trailers is used for *framing* (delimiting messages). SASL [111] or TLS/SSL [112] are used for *authentication* (verifying user identities) and *privacy* (protecting against third-party interception). For Grid applications, TLS/SSL is used within the context of the Grid Security Infrastructure (GSI) [113]. 3-digit and localized textual diagnostics are used for *reporting* (conveying status information such as errors).

We propose to encode message types and their parameters with the straightforward XML representations given in the previous Section 7.4, using the MIME type `text/xml`, which is the default in BEEP.

Finally note that it would be interesting to use SOAP [9] as a high-level tool for PDP messaging. Most commonly, HTTP 1.1 [34] is used as SOAP transport. However, SOAP is transport protocol independent. For strongly increased efficiency and low latency, SOAP should be carried over BEEP. See [31] for an IETF draft specifying a straightforward SOAP/BEEP binding.

## 7.6 Node State Table

Let us now discuss the minimum state information a node must maintain for correct operation, introduced by means of an example. Consider the state of the central grey node depicted in Figure 7.2. The node has three neighbors `(A, B, C)`. Two concurrent queries `Q1` and `Q2` are being handled. `Q1` arrives from `A` and is forwarded to `B` and `C`. `Q2` also arrives from `A`, but neighbor selection dictates that it is forwarded to `C` only. `Q1` uses three channels `(1, 2, 3)` whereas `Q2` uses two channels `(4, 5)`. The session `(a)` incoming from `A` has two channels `(1, 4)`. The session `(b)` outgoing to `B` has one channel `(2)`. The session `(c)` outgoing to `C` has two channels `(3,5)`. The sessions outgoing to `A` and incoming from `B` and `C` are not shown, because they are not used by the two queries. Both queries carry a transaction identifier, or `tid` for short: `Q1 (tid=100)` and `Q2 (tid=200)`.

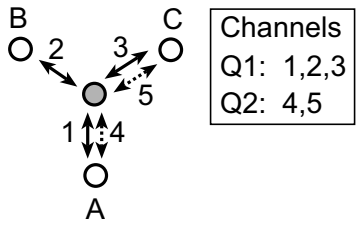


Figure 7.2: Example Node State.

For the purpose of exposition, the relational data model is used to describe state. For each session, a node must maintain at a minimum the following information:

- The channels associated with each session, so that all channels of a given session can be closed (Table 7.4 - compressed for clarity).

| Session (1) | Channel (N) |
|---|---|
| a | 1, 4 |
| b | 2 |
| c | 3, 5 |

Table 7.4: Channels of Sessions.

For each known query, a node must maintain the following information:

- The channels outgoing to dependents, so that incoming messages such as MSG RECEIVE and MSG CLOSE can be forwarded to dependents (Table 7.5).

Further, for each known query a node must maintain at a minimum the following information (Tables 7.6 and 7.7):

- The channel incoming from the client (to know where to accept messages and return responses)

| Tid (1) | Outgoing channel to dependent (N) |
|---|---|
| 100 | 2 |
| 100 | 3 |
| 200 | 5 |

Table 7.5: Channels of Queries.

- The loop timeout and abort timeout
- The current state of the transaction (`open` or `closed`)
- The execution plan to invoke upon accepting a MSG RECEIVE request
- The results currently available for immediate non-blocking delivery with the next SEND batch, as well as their number (`minAvailResults`); in practice these are implicitly contained in the execution plan
- Query scope parameters such as the maximum (and already sent) result set size as well as maximum (and already sent) byte size of the result set to be delivered, as specified by the MSG QUERY message

| Tid | Incoming Channel from Client | Abort Timeout | Loop Timeout | State | Execution Plan |
|---|---|---|---|---|---|
| 100 | 1 | 20 | 30 | Open | Plan 1 |
| 200 | 4 | 50 | 60 | Open | Plan 2 |

Table 7.6: Query State.

| Tid | AvailResults | MinAvailResults | MaxResults | SentResults | MaxResultsBytes | SentBytes |
|---|---|---|---|---|---|---|
| 100 | {r1,r2,r3} | 3 | 10 | 5 | 10000 | 5000 |
| 200 | {r4,r5} | 2 | 100 | 50 | 20000 | 10000 |

Table 7.7: Query State Continued.

For the purpose of exposition, the relational data model is used to describe state. Clearly an implementation should use efficient data structures such as hash maps for frequent lookup operations such as finding the transaction identifier of an incoming channel.

## 7.7 Related Work

**RDBMS.** The network protocols of Relational Database Management Systems are designed with a tight focus on a single node architecture and response model. For maximum efficiency, communication is tightly coupled, for example with low overhead Inter Process

Communication (IPC) mechanisms carried over more efficient layers than TCP. Like our approach, RDBMS protocols also are stateful and allow for low latency, pipelining, early and/or partial result set retrieval due to synchronous pull, and result set delivery in one or more variable sized batches. They provide very strong functionality to provide for resource consumption and flow control on a per query and/or per user basis. Low-level RDBMS interfaces such as Oracle's OCI [114] allow for application multiplexing. High-level access APIs such as JDBC [115] do not provide access to such facilities; they use less scalable TCP connection pooling instead. RDBMS protocols are closed and proprietary (except for open source products), and hence unsuitable for Internet-level interoperability and extensibility.

**LDAP and MDS.** The LDAP model allows for multi-level hierarchical topologies as well as normal and referral response modes. It does not support arbitrary topologies and direct response mode. MDS additionally supports routed response mode but otherwise has the same properties as LDAP. Like our approach, both protocols are stateful as well as connection and message-oriented. They do not support synchronous pull, and result set delivery in one or more variable sized batches. Synchronous paging behavior has been proposed [74], but this is still inefficient, because each response message still contains a single entry only. They do support asynchronous push. They do not provide for resource consumption and flow control on a per query basis. They lack a concept that concentrates all messages related to a query, like a BEEP channel[2]. LDAP has a notion of application multiplexing that is not equivalent to ours. The fact that messages may be unordered is dictated by the LDAP network protocol. The BEEP network protocol guarantees for ordered message delivery. If LDAP were used for PDP messaging, the parameters `Q` and `R` of a SEND message would be meaningless. Likewise, the server response for a CLOSE request would be allowed to "overtake" the SEND responses for prior RECEIVE requests, which violates pipelining semantics. Like BEEP, LDAP is an IETF standard.

**Gnutella and Freenet.** Gnutella and Freenet support queries in arbitrary graph topologies but only a single response mode. Like our approach, their protocols are stateful as well as connection and message-oriented. They do not support synchronous pull but they do support asynchronous push with one or more variable sized batches. Like LDAP and MDS, they do not provide for resource consumption and flow control on a per query basis. However, they do have a notion of application multiplexing that is equivalent to ours for the purpose of result *set* retrieval. Their protocol specifications are not closed and proprietary, but they are ad-hoc specifications without any relation to an open IETF standard and its implied quality in terms of interoperability and extensibility.

[2]The specification reads "Note that although servers are required to return responses whenever such responses are defined in the protocol, there is no requirement for synchronous behavior on the part of either client or server implementations: requests and responses for multiple operations may be exchanged by client and servers in any order, as long as clients eventually receive a response for every request that requires one" [14].

## 7.8 Summary

**Comparison with Related Work.** We describe how the operations of the Unified Peer-to-Peer Database Framework (UPDF) and registry `XQuery` interface from Section 5.2 are carried over (bound to) a network protocol. We develop a messaging, communication and network protocol model, collectively termed *Peer Database Protocol (PDP)*. PDP supports P2P database queries for a wide range of database architectures and response models such that the stringent demands of ubiquitous Internet infrastructures in terms of scalability, efficiency, interoperability, extensibility and reliability can be met.

PDP has a number of key properties. It is applicable to any node topology (e.g. centralized, distributed or P2P) and to multiple P2P response modes (routed response and direct response, both with and without metadata modes). To support loosely coupled autonomous Internet infrastructures, the model is connection-oriented (ordered, reliable, congestion sensitive) and message-oriented (loosely coupled, operating on structured data). For efficiency, it is stateful at the protocol level, with a transaction consisting of one or more discrete message exchanges related to the same query. It allows for low latency, pipelining, early and/or partial result set retrieval due to synchronous pull, and result set delivery in one or more variable sized batches. It is efficient, due to asynchronous push with delivery of multiple results per batch. It provides for resource consumption and flow control on a per query basis, due to the use of a distinct channel per transaction. It is scalable, due to application multiplexing, which allows for very high query concurrency and very low latency, even in the presence of secure TCP connections. To encourage interoperability and extensibility it is fully based on Internet Engineering Task Force (IETF) standards, for example in terms of asynchrony, encoding, framing, authentication, privacy and reporting.

These key properties distinguish our approach from related work, which individually addresses some, but not all of the above issues. We are not aware of related work that proposes a uniform messaging model that is applicable to any node topology and at the same time to multiple P2P response modes. Some related work does not apply to loosely coupled autonomous database nodes (RDBMS). Some protocols are not stateful at the protocol level (HTTP based mechanisms). Some do not support synchronous pull (LDAP, MDS, Gnutella, Freenet) and result set delivery in one or more variable sized batches (LDAP, MDS, HTTP based mechanisms). Some do not support asynchronous push with delivery of multiple results per batch (LDAP, MDS, HTTP based mechanisms). Some do not provide for resource consumption and flow control on a per query basis (LDAP, MDS, Gnutella, Freenet, HTTP based mechanisms). Some lack application multiplexing for scalable query concurrency (some RDBMS drivers, HTTP based mechanisms, LDAP, MDS). Some do not encourage interoperability and extensibility based on open IETF standards (RDBMS, Gnutella, Freenet).

**Summary.** The high-level messaging model employs four request messages (QUERY, RECEIVE, INVITE, CLOSE) and a response message (SEND). A *transaction* is a sequence of one or more message exchanges between two peers (nodes) for a given query. An example transaction is a QUERY-RECEIVE-SEND-RECEIVE-SEND-CLOSE sequence. A peer can concurrently handle multiple independent transactions. A transaction is identified by a transaction identifier. Every message of a given transaction carries the same transaction

identifier.

A QUERY message is forwarded along hops through the topology. A RECEIVE message is used by a client to request query results from another node. It requests the node to respond with a SEND message, containing a batch of at least $N$ and at most $M$ results from the (remainder of the) result set. A client may issue a CLOSE request to inform a node that the remaining results (if any) are no longer needed and can safely be discarded. If the local result set is not empty under direct response, the node directly contacts the agent with an INVITE message to solicit a RECEIVE message. A RECEIVE request can ask to deliver SEND messages in either synchronous (pull) or asynchronous (push) mode. In synchronous mode a single RECEIVE request must precede every single SEND response. An example sequence is RECEIVE-SEND-RECEIVE-SEND. In asynchronous mode a single RECEIVE request asks for a sequence of successive SEND responses. A client need not explicitly request more results, as they are automatically pushed in a sequence of zero or more SENDs. An example sequence is RECEIVE-SEND-SEND-SEND. Appropriately sized batched delivery greatly reduces the number of hops incurred by a single RECEIVE. To reduce latency, a node may prefetch query results.

A node maintains a state table that keeps for each query at a minimum the transaction identifier, abort timeout, loop timeout and an open/closed state flag. A query can be in three states: *unknown*, *open* or *closed*. The rules governing state transitions are detailed.

The model is mapped down to the abstract messaging model of the BEEP application level network protocol framework. The permitted kinds of message exchanges are detailed. Message types and their parameters are mapped to XML representations.

An abstract *communication model* is discussed that spells out how a message is carried from one peer to another. The BEEP communication model operates on abstract entities such as sessions, channels, messages and frames. Two peers establish a *session* for communication. Within a session, one or more *channels* can be established. A channel carries zero or more *messages*. A message can have arbitrary length and content. A message is segmented into one or more *frames* of variable length. Within a channel, *pipelining* is provided (messages are serialized). Channels are isolated from each other, and therefore handle asynchrony/multiplexing. Inter-channel messages may be unordered. In our context, o*ne channel per distinct transaction* is used to isolate communication referring to different queries and to ensure that messages of a transaction are processed in serial order. *Flow control* issues are discussed arising from the use of concurrent channels of a session.

There are two options to map TCP connections: one TCP connection per channel (*TCP multiplexing* or *TM*) and one TCP connection per session (*application multiplexing* or *AM*). The properties of both models are discussed. TM has the distinct disadvantage of being much less efficient in the presence of high frequency channel creation. Hence, application multiplexing is chosen. A node with N neighbors has N incoming and N outgoing TCP connections.

The BEEP framework is adopted because it is designed for application multiplexing and integrates existing best-of-breed standards. It defines how to handle asynchrony (handling independent exchanges), encoding (representing messages), framing (delimiting messages), authentication (verifying user identities), privacy (protecting against third-party interception) and reporting (conveying status information such as errors). The BEEP transport mapping

to TCP uses application multiplexing (one TCP connection per session).

# Chapter 8

# Conclusion

## 8.1 Summary

This thesis tackles the problems of information, resource and service discovery arising in large distributed Internet systems spanning multiple administrative domains. We show how to support expressive general-purpose queries over a view that integrates autonomous dynamic database nodes from a wide range of distributed system topologies. The work was carried out in the context of the European DataGrid project (EDG) at CERN, the European Organization for Nuclear Research, and supported by the Austrian Ministerium für Wissenschaft, Bildung und Kultur.

**Service Discovery Processing Steps.** A key question is:

- *What distinct problem areas and processing steps can be distinguished in order to enable flexible remote invocation in the context of service discovery?*

To establish the context, we outline eight problem areas and their associated processing steps, namely *description, presentation, publication, request, discovery, brokering, execution* and *control*. We propose a simple grammar (*SWSDL*) for describing network services as collections of service interfaces capable of executing operations over network protocols to endpoints. The grammar is intended to be used in the high-level architecture and design phase of a software project. A service must present its current description so that clients from anywhere can retrieve it at any time. For broad acceptance, adoption and easy integration of legacy services, an HTTP hyperlink is chosen as an identifier and retrieval mechanism (*service link*). A registry for publication and query of service and resource presence information is outlined. Reliable, predictable and simple distributed registry state maintenance in the presence of service failure, misbehavior or change is addressed by a simple and effective soft state mechanism. The notions of request, resource and operation are clarified. We outline the discovery step, which finds services implementing the operations required by a request. The brokering step determines an invocation schedule, which is a mapping over time of unbound operations to service operation invocations using given resources. The execution step implements a schedule. It uses the supported protocols to invoke operations on remote services. We discuss how one can reliably support monitoring and controlling the lifecycle of a request in the presence of a service that cannot reliably complete a request within a short and well-known expected timeframe.

**Discovery Data Model and Query Language.** The key problem is:

- *What kind of database, query and data model as well as query language can support simple and complex dynamic information discovery with as few as possible architecture and design assumptions? In particular, how can one uniformly support queries in a wide range of distributed system topologies and deployment models, while at the same time accounting for their respective characteristics?*

We develop a database and query model as well as a generic and dynamic data model that address the given problem. All subsequent chapters are based on these models. Unlike in the relational model the elements of a tuple in our data model can hold structured or semi-structured data in the form of any arbitrary well-formed XML document or fragment. An individual tuple element may, but need not, have a schema, in which case the element must be valid according to the schema. The elements of all tuples may, but need not, share a common schema. The concepts of (logical) query and (physical) query scope are cleanly separated rather than interwoven. A query is formulated against a global database view and is insensitive to link topology and deployment model. In other words, to a query the set of all tuples appears as a single homogenous database, even though the set may be (recursively) partitioned across many nodes and databases. The query scope, on the other hand, is used to navigate and prune the link topology and filter on attributes of the deployment model. A query is evaluated against a set of tuples. The set, in turn, is specified by the scope. Conceptually, the scope is the input fed to the query. Example service discovery queries are given. Three query types are identified, namely *simple, medium* and *complex*. An appropriate query language (XQuery) is suggested. The suitability of the query language is demonstrated by formulating the example prose queries in the language. Detailed requirements for a query language supporting service and resource discovery are given. The capabilities of various query languages are compared.

**Database for Discovery of Distributed Content.** The key problem is:

- *How should a database node maintain information populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources? In particular, how should it do so without sacrificing reliability, predictability and simplicity? How can powerful queries be expressed over time-sensitive dynamic information?*

A type of database is developed that addresses the problem. A database for XQueries over dynamic distributed content is designed and specified – the so-called *hyper registry*. The hyper registry has a number of key properties. An XML data model allows for structured and semi-structured data, which is important for integration of heterogeneous content. The XQuery language allows for powerful searching, which is critical for non-trivial applications. Database state maintenance is based on soft state, which enables reliable, predictable and simple content integration from a large number of autonomous distributed content providers. Content link, content cache and a hybrid pull/push communication model allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client.

**Web Service Discovery Architecture.** The key problem is:

- *Can we define a discovery architecture that promotes interoperability, embraces industry standards, and is open, modular, flexible, unified, non-intrusive and simple yet powerful?*

We propose and specify a discovery architecture, the so-called *Web Service Discovery Architecture (WSDA)*. WSDA views the Internet as a large set of services with an extensible set of well-defined interfaces. It promotes an interoperable web service layer on top of existing and future Internet software, because it defines appropriate services, interfaces, operations and protocol bindings. WSDA subsumes an array of disparate concepts, interfaces and protocols under a single semi-transparent umbrella. It specifies a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery. These primitives cover service identification, service description retrieval, data publication as well as minimal and powerful query support. The individual primitives can be combined and plugged together by specific clients and services to yield a wide range of behaviors and emerging synergies. Finally, we compare in detail the properties of WSDA with the emerging Open Grid Service Architecture.

**Unified Peer-to-Peer Database Framework.** The key problems are:

- *What are the detailed architecture and design options for P2P database searching in the context of service discovery? What response models can be used to return matching query results? How should a P2P query processor be organized? What query types can be answered (efficiently) by a P2P network? What query types have the potential to immediately start piping in (early) results? How can a maximum of results be delivered reliably within the time frame desired by a user, even if a query type does not support pipelining? How can loops be detected reliably using timeouts? How can a query scope be used to exploit topology characteristics in answering a query? For improved efficiency, how can queries be executed in containers that concentrate distributed P2P database nodes into hosting environments with virtual nodes?*

- *Can we devise a unified P2P database framework for general-purpose query support in large heterogeneous distributed systems spanning many administrative domains? More precisely, can we devise a framework that is unified in the sense that it allows to express specific applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options?*

We take the first steps towards unifying the fields of database management systems and P2P computing, which so far have received considerable, but separate, attention. We extend database concepts and practice to cover P2P search. Similarly, we extend P2P concepts and practice to support powerful general-purpose query languages such as XQuery and SQL. As a result, we propose the so-called *Unified Peer-toPeer Database Framework (UPDF)* for general-purpose query support in large heterogeneous distributed systems spanning many

administrative domains. UPDF is unified in the sense that it allows to express specific applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options.

**Unified Peer-to-Peer Database Protocol.** The key problem is:

- *What messaging and communication model, as well as network protocol, uniformly support P2P database queries for a wide range of database architectures and response models such that the stringent demands of ubiquitous Internet discovery infrastructures in terms of scalability, efficiency, interoperability, extensibility and reliability can be met? In particular, how can one allow for high concurrency, low latency as well as early and/or partial result set retrieval? How can one encourage resource consumption and flow control on a per query basis?*

These problems are addressed by developing a suitable messaging, communication and network protocol model, collectively termed *Peer Database Protocol (PDP)*. PDP describes how the operations of the Peer-to-Peer Database Framework (UPDF) and registry `XQuery` interface are carried over (bound to) a network protocol. PDP has a number of key properties. It is applicable to any node topology (e.g. centralized, distributed or P2P) and to multiple P2P response modes (routed response and direct response, both with and without metadata modes). To support loosely coupled autonomous Internet infrastructures, the model is connection-oriented (ordered, reliable, congestion sensitive) and message-oriented (loosely coupled, operating on structured data). For efficiency, it is stateful at the protocol level, with a transaction consisting of one or more discrete message exchanges related to the same query. It allows for low latency, pipelining, early and/or partial result set retrieval due to synchronous pull, and result set delivery in one or more variable sized batches. It is efficient, due to asynchronous push with delivery of multiple results per batch. It provides for resource consumption and flow control on a per query basis, due to the use of a distinct channel per transaction. It is scalable, due to application multiplexing, which allows for very high query concurrency and very low latency, even in the presence of secure TCP connections. To encourage interoperability and extensibility it is fully based on Internet Engineering Task Force (IETF) standards, for example in terms of asynchrony, encoding, framing, authentication, privacy and reporting.

## 8.2 Directions for Future Research

The results presented in this thesis open five interesting research directions.

*First*, it would be interesting to extend further the unification and extension of concepts from Database Management Systems and P2P computing. For example, one could consider the application of database techniques such as buffer cache maintenance, view materialization, placement and selection as well as query optimization for use in P2P computing. These techniques would need to be extended in the light of the complexities stemming from autonomous administrative domains, inconsistent and incomplete (soft) state, dynamic and flexible cache freshness policies and, of course, tuple updates. An important problem left

open in our work is the question if a query processor can automatically determine whether a correct merge query and unionizer exist, and if so, how to choose them. Here approaches from query rewriting for heterogeneous and homogenous relational database systems [45, 95] should prove useful. Further, database resource management and authorization mechanisms might be worthwhile to consider for specific flow control policies per query or per user.

*Second*, it would be interesting to study and specify in more detail specific cache freshness interaction policies between content provider, hyper registry and client (query). Our specification allows expressing a wide range of policies, some of which we outline, but we do not evaluate in detail the merits and drawbacks of any given policy.

*Third*, it would be valuable to rigourously assess, review and compare the Web Service Discovery Architecture (WSDA) and the Open Grid Service Architecture (OGSA) in terms of concepts, design and specifications. A strong goal is to achieve convergence by extracting best-of-breed solutions from both proposals. Future collaborative work could further improve current solutions, for example in terms of simplicity, orthogonality and expressiveness. For practical purposes, our pedagogical service description language (SWSDL) could be mapped to WSDL, taking into account the OGSA proposal. This would allow to use SWSDL as a tool for greatly improved clarity in high-level architecture and design discussions, while at the same time allowing for painstakingly detailed WSDL specifications addressing ambiguity and interoperability concerns.

*Fourth*, Tim Berners-Lee designed the World Wide Web as a consistent interface to a flexible and changing heterogeneous information space for use by CERN's staff, the High Energy Physics community, and, of course, the world at large. The WWW architecture [86] rests on four simple and orthogonal pillars: URIs as identifiers, HTTP for retrieval of content pointed to by identifiers, MIME for flexible content encoding, and HTML as the primus-inter-pares (MIME) content type. Based on our Dynamic Data Model (DDM), we hope to proceed further towards a self-describing meta content type that retains and wraps all four WWW pillars "as is", yet allows for flexible extensions in terms of identification, retrieval and caching of content. Judicious combination of the four Web pillars, DDM, the Web Service Discovery Architecture (WSDA), the Hyper Registry, the Unified Peer-to-Peer Database Framework (UPDF) and its associated Peer Database Protocol (PDP) are used to define how to bootstrap, query and publish to a dynamic and heterogeneous information space maintained by self-describing network interfaces.

*Fifth*, we are starting to build a system prototype with the aim of reporting on experience gained from application to an existing large distributed system such as the European DataGrid.

# Chapter 9

# Acknowledgements

This thesis is dedicated to my family. Dietlind and Gert put me on the trajectory that eventually led to this thesis. Dietlind has a way to spread fairness and humanity. She is much engaged in social life, and draws satisfaction from interpretation of symbols from the history of arts such as the *raven*. Gert constantly radiates perspectives on all aspects of life. He induced the sense of creation, taught us evolution, the history of earth and concentration camps, and what science was about. He showed how to walk off track, making us see and understand rock faces, the pull of mountains and a wild will for freedom. Thanks to my brothers for everything we shared. Stefan, the mountaineer and straightforward no-nonsense physician in remote Northern India, will, with luck, soon be based in Mongolia. If you need his treatment, watch out for martial disinfections. Ulli and Renate favour handling snowboards and skateboards, cameras, extravagant body and hair designs, and, of course, their love.

This piece of paper is also prominently dedicated to Ben Segal, who led the CERN team, day in day out supporting this work with great integrity, enthusiasm and, above all else, in a distinct spirit of humanity and friendship. Frankly, this work would not exist without you, Ben. Gerti Kappel, Erich Schikuta and Bernd Panzer-Steindel patiently advised and guided this thesis, suggesting what always turned out to be wise alleys.

I am happy to remember Brian Tierney who shared office with me in Geneva for the last year. We had a good time hiking in the mountains of Chamonix and the Jura. With the wine and food prepared by Christine's magic hands, we'd get ready for late evening discussions. Sorry you both had to return to Berkeley. Thanks to Annemarie for having appeared at the right moment, at the right place, and also for introducing the cembalos, flutists and counter tenors of Vienna. Thanks to Kurt Stockinger, through many CERN years the spare time friend, co-worker and Mr. Query Optimizer, as well as Heinz Stockinger.

Testing ideas against the solid background of Dirk Düllmann, Ian Foster, Johannes Gutleber, Koen Holtman and Carl Kesselman proved an invaluable recipe in separating wheat from chaff. All of the above, as well as German Cancio, Francesco Giacomini, Leanne Guy, Peter Kunszt, Javier Jaen-Martinez, Gavin McCance, Asad Samar and many more colleagues from the CERN IT Division, DataGrid Architecture Task Force, DataGrid, Globus, RD45 and CMS collaborations engaged in countless discussions, contributing thoughtful perspectives. Wolfgang von Rüden and Les Robertson provided encouragement and essential organizational backing. Dominique Dupraz cared for ceremonial wine and cake and the rituals on the hallway of the department. Karlheinz Schindl, Werner Jank and Chris Fabjan kindly assisted in navigating the Austrian Doctoral Student Programme and the CERN fellowship.

# References


[1] Ben Segal. Grid Computing: The European Data Grid Project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, France, October 2000.

[2] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *1st IEEE/ACM Int. Workshop on Grid Computing (Grid'2000)*, Bangalore, India, December 2000.

[3] Dirk Düllmann, Wolfgang Hoschek, Javier Jean-Martinez, Asad Samar, Ben Segal, Heinz Stockinger, and Kurt Stockinger. Models for Replica Synchronisation and Consistency in a Data Grid. In *10th IEEE Symposium on High Performance and Distributed Computing (HPDC-10)*, San Francisco, California, August 2001.

[4] Large Hadron Collider Committee. Report of the LHC Computing Review. Technical report, CERN/LHCC/2001-004, April 2001. http://lhc-computing-review-public.web.cern.ch/lhc-computing-review-public/Public/Report_final.PDF.

[5] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. Journal of Supercomputer Applications*, 15(3), 2001.

[6] Ian Foster, Carl Kesselman, Jeffrey Nick, and Steve Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, January 2002.

[7] P. Cauldwell, R. Chawla, Vivek Chopra, Gary Damschen, Chris Dix, Tony Hong, Francis Norton, Uche Ogbuji, Glenn Olander, Mark A. Richman, Kristy Saunders, and Zoran Zaev. *Professional XML Web Services*. Wrox Press, 2001.

[8] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. *W3C Note 15*, 2001. www.w3.org/TR/wsdl.

[9] World Wide Web Consortium. Simple Object Access Protocol (SOAP) 1.1. *W3C Note 8*, 2000.

[10] UDDI Consortium. UDDI: Universal Description, Discovery and Integration. www.uddi.org.

[11] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. *W3C Recommendation*, October 2000.

[12] World Wide Web Consortium. XML Schema Part 0: Primer. *W3C Recommendation*, May 2001.

[13] International Telecommunications Union. Recommendation X.500, Information technology – Open System Interconnection – The directory: Overview of concepts, models, and services. *ITU-T*, November 1995.

[14] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. *IETF RFC 1777*, March 1995.

[15] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In *Tenth IEEE Int. Symposium on High-Performance Distributed Computing (HPDC-10)*, San Francisco, California, August 2001.

[16] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *6th Int. Symposium on High Performance Distributed Computing (HPDC '97)*, 1997.

[17] Steven Tuecke, Karl Czajkowski, Ian Foster, Jeffrey Frey, Steve Graham, and Carl Kesselman. Grid Service Specification, February 2002.

[18] World Wide Web Consortium. XQuery 1.0: An XML Query Language. *W3C Working Draft*, December 2001.

[19] International Organization for Standardization (ISO). Information Technology-Database Language SQL. *Standard No. ISO/IEC 9075:1999*, 1999.

[20] P. Mockapetris. Domain Names - Implementation and Specification. *IETF RFC 1035*, November 1987.

[21] Gnutella Community. Gnutella Protocol Specification v0.4. dss.clip2.com/GnutellaProtocol04.pdf.

[22] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[23] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical report, U.C. Berkeley UCB//CSD-01-1141, 2001.

[24] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

[25] M. van Steen, P. Homburg, and A. Tanenbaum. A wide-area distributed system. *IEEE Concurrency*, 1999.

[26] Steven E. Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony D. Joseph, and Randy Katz. An Architecture for a Secure Service Discovery Service. In *Fifth Annual Int. Conf. on Mobile Computing and Networks (MobiCOM '99)*, Seattle, WA, August 1999.

[27] Beverly Yang and Hector Garcia-Molina. Efficient Search in Peer-to-Peer Networks. In *22nd Int. Conf. on Distributed Computing Systems*, Vienna, Austria, July 2002.

[28] Adriana Iamnitchi and Ian Foster. On Fully Decentralized Resource Discovery in Grid Environments. In *Int. IEEE Workshop on Grid Computing*, Denver, Colorado, November 2001.

[29] A. Puniyani B. Huberman L. Adamic, R. Lukose. Search in power-law networks. *Phys. Rev*, E(64), 2001.

[30] S. Bradner. Key Words for use in RFCs to Indicate Requirement Levels. *IETF RFC 2119*, March 1997.

[31] E. O'Tuathail and M. Rose. Using SOAP in BEEP. *IETF Draft draft-etal-beep-soap-06*, January 2002.

[32] J. Postel and J. Reynolds. File Transfer Protocol (FTP). *IETF RFC 959*, October 1985.

[33] J. Postel. Using SOAP in BEEP. *IETF RFC 821*, August 1982.

[34] R. Fielding, J. Gettys, J.C. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. *IETF RFC 2616*. UC Irvine, Digital Equipment Corporation, MIT.

[35] Marshall Rose. The Blocks Extensible Exchange Protocol Core. *IETF RFC 3080*, March 2001.

[36] Marshall Rose. Mapping the BEEP Core onto TCP. *IETF RFC 3081*, March 2001.

[37] S. Gullapalli, K. Czajkowski, C. Kesselman, and S. Fitzgerald. The grid notification framework. Technical report, Grid Forum Working Draft GWD-GIS-019, June 2001. http://www.gridforum.org.

[38] Rajkumar Buyya, Steve Chapin, and David DiNucci. Architectural Models for Resource Management in the Grid. In *1st IEEE/ACM Int. Workshop on Grid Computing (GRID 2000)*, Bangalore, India, December 2000.

[39] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *7th IEEE Int. Symposium on High Performance Distributed Computing (HPDC'98)*, Chicago, IL, July 1998.

[40] I. Foster, A. Roy, and V. Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *8th Int. Workshop on Quality of Service*, 2000.

[41] Nelson Minar. Peer-to-Peer is Not Always Decentralized. In *The O'Reilly Peer-to-Peer and Web Services Conference*, Washington, D.C., November 2001.

[42] J.D. Ullman. Information integration using logical views. In *Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.

[43] Daniela Florescu, Ioana Manolescu, Donald Kossmann, and Florian Xhumari. Agora: Living with XML and Relational. In *Int. Conf. on Very Large Data Bases (VLDB)*, Cairo, Egypt, February 2000.

[44] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):808–823, 1998.

[45] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, September 2000.

[46] Dan Suciu. On Database Theory and XML. *SIGMOD Record*, 30(3), 2001.

[47] Mary Fernandez, Morishima Atsuyuki, Dan Suciu, and Tan Wang-Chiew. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin*, 24(2), 2001.

[48] Daniela Florescu, Ioana Manolescu, and Donald Kossmann. Answering XML Queries over Heterogeneous Data Sources. In *Int. Conf. on Very Large Data Bases (VLDB)*, Roma, Italy, September 2001.

[49] Brian Tierney, Ruth Aydt, Dan Gunter, Warren Smith, Valerie Taylor, Rich Wolski, and Martin Swany. A Grid Monitoring Architecture. Technical report, Grid Forum Working Draft GWD-Perf-16-2, January 2002. http://www.gridforum.org.

[50] World Wide Web Consortium. XML Query Use Cases. *W3C Working Draft*, December 2001.

[51] World Wide Web Consortium. XML Query Requirements. *W3C Working Draft*, February 2001.

[52] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Functions and Operators. *W3C Working Draft*, December 2001.

[53] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: an XML Query Language for Heterogeneous Data Sources. *Lecture Notes in Computer Science*, 42(7), December 2000.

[54] World Wide Web Consortium. XML Path Language (XPath) Version 1.0. *W3C Recommendation*, November 1999.

[55] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). In *The Query Languages Workshop (QL'98)*, Boston, Massachussets, December 1998.

[56] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *Eighth Int. World Wide Web Conference*, 1999.

[57] Rick Cattell et al. *The Object Database Standard: ODMG-93, Release 1.2.* Morgan Kaufmann Publishers, San Francisco, 1996.

[58] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Data Model. *W3C Working Draft*, December 2001.

[59] World Wide Web Consortium. XSL Transformations (XSLT) 2.0. *W3C Working Draft.*

[60] World Wide Web Consortium. XML Pointer Language (XPointer). *W3C Last Call Working Draft*, January 2001.

[61] Steve Fisher et al. Information and Monitoring (WP3) Architecture Report. Technical report, DataGrid-03-D3.2, January 2001.

[62] W. P. Dinda and B. Plale. A Unified Relational Approach to Grid Information Services. Technical report, Grid Forum Informational Draft GWD-GIS-012-1, February 2001. http://www.gridforum.org.

[63] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, July 1999.

[64] Apache Software Foundation. The Apache HTTP Server. http://httpd.apache.org.

[65] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. *IETF RFC 2396.*

[66] Apache Software Foundation. The Jakarta Tomcat Project. http://jakarta.apache.org/tomcat/.

[67] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. *IETF RFC 2045*, November 1996.

[68] World Wide Web Consortium. XML-Signature Syntax and Processing. *W3C Recommendation*, February 2002.

[69] P. Brittenham. An Overview of the Web Services Inspection Language, 2001. www.ibm.com/developerworks/webservices/library/ws-wsilover.

[70] Software AG. The Quip XQuery processor. http://www.softwareag.com/developer/quip/.

[71] J. Wang. A survey of web caching schemes for the Internet. *ACM Computer Communication Reviews*, 29(5), October 1999.

[72] D. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. *IETF RFC 1305*, March 1992.

[73] Dan Gunter, Brian Tierney, and Ruth Aydt. Timestamp model for grid computing. Technical report, Global Grid Forum Working Draft GWD-PERF-014-1, 2001. http://www.gridforum.org.

[74] C. Weider, A. Herron, A. Anantha, and T. Howes. LDAP Control Extension for Simple Paged Results Manipulation. *IETF RFC 2696*.

[75] M. P. Maher and C. Perkins. Session Announcement Protocol: Version 2. *IETF Internet Draft draft-ietf-mmusic-sap-v2-00.txt*, November 1998.

[76] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. Dias. Design and performance of Web server accelerator. In *Proceedings of Infocom'99*, 1999.

[77] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic Web data. In *Proceedings of Infocom'99*, 1999.

[78] The OpenLDAP project. The OpenLDAP project. http://www.openldap.org.

[79] A. Wu, H. Wang, and D. Wilkins. Performance Comparison of Web-To-Database Applications. In *Proceedings of the Southern Conference on Computing*, The University of Southern Mississippi, October 2000.

[80] S.Dar, M.J. Franklin, B. Jnnson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India, 1996.

[81] L. Chen, E.A. Rundensteiner, and S. Wang. XCache - A Semantic Caching System for XML Queries. In *ACM SIGMOD Conf. On Management of Data*, 2002. Software System Demonstration Paper.

[82] Oracle. J2EE and Microsoft .NET, April 2002. Oracle Corp., White Paper.

[83] Madhusudhan Govindara, Aleksander Slominski, Venkatesh Choppella, Randall Bramley, and Dennis Gannon. Requirements for and Evaluation of RMI Protocols for Scientific Computing. In *Supercomputing Conference (SC'00)*, Dallas, Texas, November 2000.

[84] David Culler, Kim Keeton, Lok Tim Liu, Alan Mainwaring, Rich Martin, Steve Rodrigues, Kristin Wright, and Chad Yoshikawa. The Generic Active Message Interface Specification, August 1994. Computer Science Division, University of California at Berkeley, White Paper.

[85] Maite Barroso. Grid Fabric Management Work Package Report on Current Technology. Technical report, DataGrid-04-TED-0101, May 2001.

[86] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD Thesis, University of California, Irvine, 2000.

[87] G. Panduragan, P. Raghavan, and E. Upfal. Protocols for building low diameter peer-to-peer networks. In *DIMACS Workshop on Internet and WWW Measurement, Mapping and Modeling*, Piscataway, NJ, February 2002.

[88] Ron Rivest. The MD5 message-digest algorithm. *IETF RFC 1321*, April 1992.

[89] National Institute of Standards and Technology. Secure Hash Standard. Technical report, FIPS 180-1, Washington, D.C., April 1995.

[90] Matei Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. In *Int. Conf. on Peer-to-Peer Computing (P2P2001)*, Linkoping, Sweden, August 2001.

[91] Clip2Report. Gnutella: To the Bandwidth Barrier and Beyond. http://www.clip2.com/gnutella.html.

[92] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke. A Community Authorization Service for Group Collaboration. In *IEEE 3rd Int. Workshop on Policies for Distributed Systems and Networks (submitted)*, 2001.

[93] Stefano Ceri and Giuseppe Pelagatti. *Distributed Databases - Principles and Systems*. McGraw-Hill Computer Science Series, 1985.

[94] M. Franklin, B. Jonsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *ACM SIGMOD Conf. On Management of Data*, Montreal, Canada, June 1996.

[95] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *ACM SIGMOD Conf. On Management of Data*, 1999.

[96] Dan Suciu. Distributed Query Evaluation on Semistructured Data. *ACM Transactions on Database Systems*, 2002.

[97] T. Urhan and M. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. *The Very Large Database (VLDB) Journal*, 2001.

[98] Jordan Ritter. Why Gnutella Can't Scale. No, Really. http://www.tch.org/gnutella.html.

[99] S.E. Deering. *Multicast Routing in a Datagram Internetwork*. PhD Thesis, Stanford University, 1991.

[100] Java Community Process. Java Servlet 2.3 Specification. jcp.org/aboutJava/communityprocess/final/jsr053.

[101] Java Community Process. Enterprise Java Beans Specification. java.sun.com/products/ejb/docs.html.

[102] IEEE. *Data Engineering Bulletin*, 23(2), June 2000.

[103] Jayavel Shanmugasundaram, Kristin Tufte, David J. DeWitt, Jeffrey F. Naughton, and David Maier. Architecting a Network Query Engine for Producing Partial Results. In *WebDB 2000 (Informal Proceedings)*, 2000.

[104] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Integrating Network-Bound XML Data. *IEEE Data Engineering Bulletin*, 24(2), 2001.

[105] Jeffrey F. Naughton, David J. DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, and Rushan Chen. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2), 2001.

[106] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *First Int. Conf. on Parallel and Distributed Information Systems (PDIS)*, December 1991.

[107] Zachary G. Ives, Daniela Florescu, Marc T. Friedman, Alon Y. Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *ACM SIGMOD Conf. On Management of Data*, 1999.

[108] Tolga Urhan and Michael J. Franklin. Xjoin, A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2), June 2000.

[109] Tolga Urhan and Michael J. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. In *Int. Conf. on Very Large Data Bases (VLDB)*, 2001.

[110] Beepcore Community. The beepcore open source project for Java, C and Tcl. http://www.beepcore.org.

[111] J. Myers. Simple Authentication and Security Layer (SASL). *IETF RFC 2222*, October 1997.

[112] T. Dierks and C. Allen. The TLS Protocol Version 1.0. *IETF RFC 2246*, January 1999.

[113] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch. A National-Scale Authentication Infrastructure. *IEEE Computer*, 33(12), 2000.

[114] Oracle Corp. Oracle Call Interface Programmer's Guide, January 2002. Release 9.0.1, Part Number A89857-01.

[115] Donald Bales. *Java Programming with Oracle JDBC*. O'Reilly, December 2001. ISBN 0-596-00088-x.